

Charles University in Prague  
Faculty of Mathematics and Physics

## **MASTER THESIS**



Andrei Conicov

### **Indexing Linked Data**

Department of Software Engineering

Supervisor of the master thesis: Mgr. Nečaský Martin, Ph.D.

Study programme: Informatics

Specialization: Software systems

Prague 2012

Words cannot express my gratitude to my advisor Martin Nečaský. I would like to thank him for his patience and professional approach, clear and friendly advices, assistance and for opening the area of linked data to me.

I would like to thank to Tetyana Bryzhachenko for language corrections, and my colleagues for their assistance. Finally I want to thank to my family for their support in my life and studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

Prague, April 11, 2012

.....

Andrei Conicov

Název práce: Indexování Linked Data

Autor: Andrei Conicov

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Nečaský Martin, Ph.D.

Abstrakt: Rychlý vývoj Webu nabídl možnost publikovat velké množství propojených dokumentů. Každý takový dokument představuje cennou informaci. Linked Data je pojem používaný k popisu metody odhalování a propojení těchto dokumentů. I když tato metoda je stále ve zkušební fázi, zpracování všech stávajících zdrojů údajů je poměrně náročné a nejzřejmějším řešením je pokusit se je indexovat.

Tato studie se zabývá otázkami, jak navrhnout index, který bude schopný pracovat s miliony takových záznamů. Analyzuje stávající projekty a popisuje index, který může splnit požadavky. Prototyp indexu a výsledky testů nabízejí další informace o účinnosti a struktuře indexu.

Klíčová slova: Indexace, RDF, Linked Data, B+-stromy

Title: Indexing Linked Data

Author: Andrei Conicov

Department / Institute: Department of Software Engineering

Supervisor of the master thesis: Mgr. Nečaský Martin, Ph.D.

Abstract: The fast evolution of the World Wide Web has offered the possibility to publish a huge amount of linked documents. Each such document represents a valuable piece of information. Linked Data is the term used to describe a method of exposing and connecting such documents. Even if this method is still in an experimental phase, it is already hard to process all existing data sources and the most obvious solution is to try and index them.

The study addresses questions on how to design an index that will be capable to operate with millions of such entries. It analyses the existing projects and describes an index that may fulfill the requirements. The prototype implementation and the provided test results offer additional information about the index structure and effectiveness.

Keywords: Indexation, RDF, Linked Data, B+-trees

<b>1.</b>	<b>Introduction</b>	<b>10</b>
1.1.	Goals	11
1.2.	Structure of work	11
<b>2.</b>	<b>Background</b>	<b>13</b>
2.1.	RDF data	13
2.2.	Linked data analyse	14
2.3.	Architecture	15
2.4.	The RDF data model	16
2.5.	Linked data examples	16
2.6.	Indexing methods	18
<b>3.</b>	<b>Related work</b>	<b>19</b>
3.1.	Redland	19
3.2.	Lucene	22
3.3.	Jena	23
3.4.	YARS	25
3.5.	3Store	27
3.6.	Sesame	29
3.7.	Conclusion	30
<b>4.</b>	<b>The design</b>	<b>31</b>
4.1.	Supported Queries	31
4.1.1.	SELECT statement	33

4.1.2.	Optional match	34
4.1.3.	Union	35
4.1.4.	Filter	35
4.2.	Index design	35
4.2.1.	Hashing	36
4.2.2.	Full text search	37
4.2.3.	Ontologies as a dimension	37
4.2.4.	Full triple index (FullDoc)	37
4.2.5.	Data structures	39
4.2.6.	R-tree	40
4.2.6.1.	Searching	42
4.2.6.2.	Effectiveness	42
4.2.7.	Multidimensional B-trees (MDBT)	43
4.2.7.1.	Searching	44
4.2.7.2.	Effectiveness	45
4.3.	Conclusion	45
<b>5.</b>	<b>RDF index design</b>	<b>47</b>
5.1.	File structure	47
5.2.	FullDoc index	48
5.3.	Subject and object index	48
5.4.	Predicate index	49

5.5.	Addresses and data blocks	50
5.6.	B+-tree structure	53
5.7.	Empty blocks	55
5.8.	Serialization effectiveness	56
5.8.1.	Cache	56
5.9.	B+-tree effectiveness	58
5.9.1.	Memory limits	59
5.9.1.1.	Node number limit	60
5.9.1.2.	Postponed insertions	61
5.9.1.3.	LRU items	62
5.10.	RDF parser and SPARQL engine	64
<b>6.</b>	<b>RDF index testing</b>	<b>66</b>
6.1.	Dataset specification	67
6.2.	Indexation	69
6.2.1.	One million triples	70
6.2.2.	Ten millions triples	76
6.2.3.	Twenty five millions triples	82
6.2.4.	Indexation results	82
6.3.	SPARQL Queries	84
6.3.1.	Query 1	86
6.3.2.	Query 2	88

6.3.3.	Query 3	89
6.3.4.	Query 4	90
6.3.5.	Query 5	92
6.3.6.	Query 6	93
6.3.7.	Query 7	94
6.3.8.	Query 8	94
6.3.9.	Query 9	95
6.3.10.	Query 10	96
6.3.11.	Query 11	97
6.3.12.	Query 12	97
6.4.	Conclusion	98
<b>7.</b>	<b>Concluding remarks</b>	<b>100</b>
	<b>Bibliography</b>	<b>101</b>
	<b>List of figures</b>	<b>106</b>
	<b>List of graphs</b>	<b>107</b>
	<b>List of tables</b>	<b>108</b>
	<b>Appendix A – User documentation</b>	<b>109</b>
	User interface	109
	RDF indexation window	109
	Hash window	111
	SPARQL window	111
	Display index window	112
	Settings window	113



Statistics window	114
<b>Appendix B – Content of the attached DVD-ROM</b>	<b>115</b>
<b>Appendix C – Program settings</b>	<b>116</b>
<b>Appendix D – Indexation statistics</b>	<b>117</b>

# 1. Introduction

The main idea behind Linked Data is to extend the WEB, by creating typed links between data from different data sources. “The term Linked Data is used to describe a method of exposing, sharing and connecting data via dereferenceable URIs on the Web.” [1]

“Linked data is the first practical expression of the semantic Web, useful and doable today, and applicable to all forms of data.” [2] By implementing the Linked Data principles it will be possible to create machine-readable metadata, as a result, the machines will be able to understand the semantics of the information on the Web. By adding the metadata, a machine can process knowledge itself, instead of text, using processes similar to human deductive reasoning and inference.

The HTML is used to describe how data will be presented to a human, but it does not explain the full meaning of the presented information, this is the part that is done by the Semantic HTML. One such example is the price comparison websites that have implemented the idea of adding meaning (metadata) to the presented information. However, the semantic web takes it even further. It involves publishing the information in languages specifically designed for data. The content described in such a manner manifests itself as descriptive data stored in Web-accessible database, or as a mark-up within documents (XHTML).

Currently this method is applied experimentally, but already the amount of such kind of data is huge. It becomes hard to process all data sources and the most obvious solution is to try and index them. The indexation process is not new and it has been used in many different cases, but it is necessary to understand that the indexation process differs according to the kind of data that is indexed. It is necessary to take into account the structure and the most frequent use cases. Of course it is possible to use one of the existing RDBMS and try to create an index that has been already implemented and tested. The question is if the existing index is good enough. In this context it is necessary to take in consideration: the time required to create the index, the speed when searching for specific data and the index size. Linked data are well formatted data that have a strictly predefined structure and

meaning; this means that the existing indexes used in RDBMS are not necessary the best and it is necessary to analyse other possibilities.

## 1.1. Goals

The goal of our work is to create a linked data (RDF) index that will be able to compete with the existing known indexes, which will be described later. It is very important to explain in detail the index structure so that it can be adjusted for specific scenarios. The proposed solution should be effective both at indexing and at answering queries. The index has to be able to operate with files that do not fit into a normal PC RAM (2 GB) this means that the files may contain more than 10 million triples and what is more important it has to be effective on such computers. Each decision that will influence the index design has to be discussed and properly explained. This work will not implement any new RDF file parsers or SPARQL query engines.

## 1.2. Structure of work

The rest of the paper is organized as follows. In chapter 2 we introduce the concept of linked data and we define the terms that are related to it. The chapter has the task to prepare the background that is necessary in order to understand the problem and the way it may be solved.

In chapter 3 we describe existing projects that have already proved their effectiveness. Also we try to analyse projects that are not so effective but however, offer interesting information from the field of research.

In chapter 4 we describe the theoretical part of the project. In this part we describe the SPARQL query language and a few concepts and data structures that may be useful later. In the end of this chapter we summarise and describe in theory the structure of our index.

Chapter 5 contains the program implementation aspects. This part offers the description of the used data structures and the most important implementation

details. Also it contains the solution of the problems that have occurred during the implementation phase, which have not been discussed in other chapters.

Chapter 6 presents the performance evaluation of our index. The tests have the objective to compare our index with existing indexes. This chapter describes the advantages and disadvantages of our index.

Chapter 7 contains concluding remarks.

## 2. Background

### 2.1. RDF data

Before analysing the existing projects that try to index the Linked Data we will explain the frequently used terms. This chapter is meant to briefly introduce the two research areas this work is addressing: the linked data and the indexation process.

First of all it is necessary to understand what RDF stands for. RDF is the abbreviation for the resource description framework; in the W3 specifications it is originally used as a metadata data model. It is based upon the idea of making statements about resources in the form of subject-predicate-object expressions. These expressions are known as triples: subject, predicate and object. The predicate denotes features of the resource and creates a relationship between the subject and the object. The subject is a URI or a blank node, the predicate is a URI and the object is a: URI, blank node or a Unicode string literal. In order to serialise data using the RDF standard it is possible to use one of the multiple serialization formats. The two most frequently used are:

- RDF/XML – XML serialization format of RDF models, also called RDF.
- Notation 3 – non-XML serialization of RDF models, also called N3, an example is presented further. It is easier to write, and in some cases easier to follow. N3 is closely related to the Turtle and N-Triples formats.

```
andreas.rdf:
<#andreas> <foaf:knows> <#michael>.
<#andreas> <foaf:knows> <#stefan>.
<#andreas> <foaf:name> "Andreas Harth".
<#andreas> <rdf:type> <foaf:Person>.
<#stefan> <rdf:type> <foaf:Person>.
```

Now that we know how RDF triples are serialised it will be easier to understand how an RDF graph looks like. The statements may be abstractly viewed as a graph. In a RDF graph the subjects and objects are the vertices, while the edges are the predicates.

Next important terms are the RDF schema and ontologies. *The RDF Schema* is an extensible knowledge representation language, which is providing basic

elements for the description of ontologies. Ontology is a “formal, explicit specification of a shared conceptualisation” [3]. Ontology provides a shared vocabulary, allowing creating marked up documents that can be processed by agents. We will not be providing any support for the ontologies, but this term may be mentioned in the projects that we will analyse later.

In order to filter the RDF data, a special SQL –like language has been designed, which is called SPARQL. Later we will present some SPARQL queries and we will explain how they are evaluated.

The last term that we would like to explain is stemming, which is the process of reducing inflected (or sometimes derived) words to their stem, base or root form – generally a written word form. We will not be implementing this technique, but it is mentioned in some of the analysed projects.

## 2.2. Linked data analyse

Indexation requires a detailed analysis of the data that are going to be indexed. It is important to understand two aspects, the first is the data structure and the second is the way it will be used. However, at first we would like to explain what is linked data. It can be little confusing but the term linked data has two meanings, one as a publication model and the second as the data itself. Linked data is considered to be a publication model, used for publishing structured data on the web. This model can be characterised as a list of principles and recommendations. Nearly every article about linked data references to the four principles that Tim Berners-Lee has enumerated:

1. Use URI to identify things. Such identification enables interaction with representations of the resource over a network using specific protocols. It is important to specify an URI that will not change with time, otherwise all links that are using this URI will have to be updated. However the same item may be identified by multiple URIs.
2. Use HTTP URIs so that these things can be referred to and looked up by people and user agents. The HTTP name lookup is a complex and a powerful set of standards that is still evolving and it is a mistake not to use it.

3. Provide useful information about the thing when its URI is dereferenced , using standard formats such as RDF/XML.
4. Include links to other (external), related URIs in the exposed data to improve discovery of the related information on the web. In this way we will create the unbounded web in which it will be possible to find all kind of things.

It is important to remember these four principles, in order not to miss the opportunity to make the data interconnected. Later the lack of interconnection may limit the way the information can be used. “It is the unexpected re-use of information which is the value added by the web” [4].

Linked data is also used as the term that defines the data that are published using this model. These structured data are interlinked offering the Web of Data or Semantic Web. The Web of Data can be accessed using special browsers. These browsers enable users to navigate between different data sources by following links. A link simply states that one piece of data has some kind of relationship to another piece of data.

Compared with other structured data that are available on the Web through Web 2.0 APIs such as eBay, Amazon or Yahoo, Linked Data has the advantage of providing a single, standardized access mechanism. This allows data sources to be:

- More easily crawled by search engines,
- Accessed using data browsers, and
- Enables links between data from different data sources.

Almost all linked data is currently expressed in RDF, as a result very often these two terms may be used as synonyms [5].

## 2.3. Architecture

Linked Data resources are identified using HTTP URIs. This notation offers a simple way to create unique names without centralized management and after dereferencing the HTTP URI it is possible to access the resource over the Web. In this context we have to mention, that the resources may be of information and non-information types.

## 2.4. The RDF data model

Linked data is usually published using the Resource Description Framework (RDF). This is simple but also strict data model. The triples may be of two main types' literals or RDF links. The literals are strings, numbers, or dates and are used as the object, which describes the properties of the resources. The RDF links are used to interlink resources. The URI in the subject and object identify the resources, while the URI in the predicate identifies the type of relation. In the example that is presented further the subject is presented by the `http://www.w3.org/People/Berners-Lee/card#i` RDF link. The relations are described by the predicates *owl:sameAs* and *foaf:knows*, that are contained in the *owl* and *foaf* namespaces.

```
# RDF links taken from Tim Berners-Lee's FOAF profile
<http://www.w3.org/People/Berners-Lee/card#i>
owl:sameAs <http://dbpedia.org/resource/Tim_Berners-Lee> ;
foaf:knows <http://www.w3.org/People/Connolly/#me> .
```

The main benefits of using the RDF data model in a Linked Data context are:

- Clients can look up every URI in an RDF graph over the Web to retrieve additional information.
- Information from different sources merges naturally.
- The data model enables you to set RDF links between data from different sources.
- The data model allows you to represent information that is expressed using different schemata in a single model.

Combined with schema languages such as RDF-S or OWL, the data model allows us to use as much or as little structure as we need, meaning that we can represent tightly structured data as well as semi-structured data.

## 2.5. Linked data examples

It has been already explained in which form linked data is published. By following the link <http://esw.w3.org/DataSetRDFDumps> it is possible to download some of the publicly available linked datasets. Further we will present the extracts from some of the frequently mentioned RDF datasets. The Lexvo dataset was used as



the testing dataset during the implementation phase. It has 350 thousands of triples which if compared to our target datasets, is a relatively small dataset.

- **DBpedia - Short Abstracts**

```
<http://dbpedia.org/resource/-30-_%28film%29>
<http://www.w3.org/2000/01/rdf-schema#comment> "-30- (released as
Deadline Midnight in the UK) is a 1959 movie starring William Conrad
and Jack Webb as the editor and publisher, respectively, of a
fictional Los Angeles evening newspaper. As the shift of a typical
day starts, in which they don't know what will happen, the newspaper
is created before our eyes as different stories are discovered and
reported."@en .
```

- **DBpedia - Pagelinks**

```
<http://dbpedia.org/resource/%21%21>
<http://dbpedia.org/property/wikilink>
<http://dbpedia.org/resource/exclamation_mark> .
```

- **Entrez Gene**

```
<owl:Thing rdf:about="http://purl.org/commons/record/ncbi_gene/1">
<dc:identifier rdf:datatype="&xsd:string">1</dc:identifier>
<dc:title rdf:datatype="&xsd:string">A1B</dc:title>
<dc:title rdf:datatype="&xsd:string">alpha-1-B
glycoprotein</dc:title>
<sciencecommons:from_species_described_by
rdf:resource="http://purl.org/commons/record/taxon/9606"/>
</owl:Thing>
```

- **Lexvo**

```
<rdf:Description rdf:about="http://lexvo.org/id/iso639-3/aad">
<rdfs:comment xml:lang="en" rdf:datatype="xsd:string">Amal is a
language of Papua New Guinea.</rdfs:comment>
<rdfs:label xml:lang="br"
rdf:datatype="xsd:string">Amaleg</rdfs:label>
<rdfs:label xml:lang="en"
rdf:datatype="xsd:string">Amal</rdfs:label>
<rdfs:label xml:lang="hr" rdf:datatype="xsd:string">Amal
jezik</rdfs:label>
<owl:sameAs rdf:resource="http://dbpedia.org/resource/Amal_language"
/>
<owl:sameAs
rdf:resource="http://www.mpii.de/yago/resource/Amal_language" />
<lvont:iso639P3PCode
rdf:datatype="xsd:string">aad</lvont:iso639P3PCode>
</rdf:Description>
```

We may notice that the type of contained information in these fragments is very different. Some of the fragments, for example Lexvo, contain short text values, while DBpedia contains a few rows long text. Some of the elements contain only the URI while others contain additional information, such as the data type and the language.

## 2.6. Indexing methods

The purpose of creating and storing an index is to optimize speed and performance in finding relevant documents for a search query. Without an index, the search engine would scan every document in the corpus, which would require considerable time and computing power. For example, while an index of 10000 documents can be queried within milliseconds, a sequential scan of every word in 10000 large documents could take hours. The additional computer storage required to store the index, as well as the considerable increase in the time required for an update to take place, are traded off for the time saved during information retrieval. To index means to extract specific information from data and to access data through it. Major factors in designing an index include:

1. Storage techniques - How to store the index data, that is, whether information should be data compressed or filtered;
2. Index size - How much disk space and RAM is required to store and to use the index.
3. Lookup speed - How quickly a word can be found in the index. The speed of finding an entry in a data structure, compared with how quickly it can be updated or removed, is a central focus of computer science.
4. Maintenance - How the index is maintained over time
5. Fault tolerance - How important it is for the service to be reliable. Issues include dealing with index corruption.

In our project we will focus on the first three factors. The index will contain triples that are not very often modified; as a result it may be considered to be read-only.

### 3. Related work

*“Work on RDF is constantly in danger of reinventing the wheel because existing work in database is ignored”. [6]*

In the following chapter we will analyse the existing projects that index RDF documents. This part of the paper is very important because it will offer us an overview of the work that has already been done in this field. It is necessary to be aware of the existing projects in order not to make the same mistakes or even maybe to find inspiration. We will try to analyse the most known projects: Redland, Lucene, YARS, Jena, 3Store and Sesame. Of course there are other projects that use indexed RDF documents, such as: Sindice, SWSE, Parka, Swoogle etc., but after a short analyse, which we will not present in this paper because of lack of space, it was clear that they will not have a big impact on our index structure. One of the reasons is the fact that they do not directly solve the indexation problem. Many of them use indexes created by other projects. For example Swoogle uses the Lucene-index and the MySQL database. Another reason is the ineffectiveness. The Parka project has an upper limit on its knowledge base size of around 2.5 million triples, which appears to be due to the structure of the relational indexes.

#### 3.1. Redland

Redland [7] [8] is one of the oldest projects, that has been implemented in the early 2000. It uses a modular approach, just as many other newer projects. During the indexation process, the triples are transformed into hashes, which are stored as pairs (key, value) in one of the supported storages. The storages may be divided into two categories:

1. In-memory - based on a custom hash implementation. Also exists a double linked list implementation, that was used to verify the correctness, but it has an  $O(n)$  search speed due to no indexing, This storage initially formed the basis used for the unit tests.

2. Serialized - using Sleepycat/Berkeley DB <sup>1</sup>, 3Store, MySQL, PostgreSQL or SQLite. This type of storage was developed to provide a more sophisticated storage solution.

This means that an in-memory store, an in-memory indexed store or a persistent indexed store are all possible.

The on-disk store uses either entirely standalone resources or widely available and stable databases. The first supported external storage was dependent on an early work on persistent B-tree-based storage Berkeley DB (BDB). The BDB approach generates very large files since the full triples including full URIs and literals are in each triple. This method is simple and has been proved to be fast enough for up to low millions of triples [7]. Later they have added support of the 3Store triple-store, but according to authors it is not clear if it will be supported in future because most of the 3store development is currently focused in other direction. Another interesting reason is the fact that 3store was developed after rejecting Redland as the optimal RDF storage for other semantic web tools[8]. The 3Store project will be analysed in next chapters.

Another supported storage methods use the well known relational databases MySQL, PostgreSQL and SQLite, but no additional information about this storages could be found. Taking into account the pair (key, value) structure, we suppose that the implemented database tables have as well a simple structure.

The following table shows the used hashes. According to the documentation, it is not necessary to create all the hashes, but in many cases this will influence the query evaluation speed.

---

<sup>1</sup> BDB stores arbitrary key/data pairs as byte arrays, and supports multiple data items for a single key. Berkeley DB is not a relational database. BDB can support thousands of simultaneous threads of control or concurrent processes manipulating databases as large as 256 terabytes, on a wide variety of operating systems including most Unix-like and Windows systems, and real-time operating systems.

Hash	Key	Value
SP2O	Subject and predicate	Object
PO2S	Predicate and object	Subject
SO2P	Subject and Object	Predicate
P2SO	Predicate	Subject and Object

Figure 1 Redland list of used hashes

When defining the structure of the hash, the key factor was the fact that the disk access is much slower than the processor time. According to the project developers it is more expensive and complex to have multiple hashes, identify the key and values uniquely and assign short IDs, than just store all of the triples in each hash.

When a query is performed using the triples matching style<sup>2</sup>, Redland picks the most appropriate hash if it is available. When it is not, Redland lists all the (key-value) pairs of one of the hashes that store all S (subjects), P (predicates) and O (objects) and uses the resulting set of triples to match against. In this way, Redland can perform queries when the hash is not available without user interaction. This store model can thus work with 1 hash only, since the S, P and O are stored in all of the hashes (1-4). However this kind of sequential search is too slow and unacceptable when dealing with a huge amount of data.

The hashes are used both for the statement queries and the node centric ones. The former are provided by serialising the hash and filtering via the querying statement. This can be very slow for large models so the node-centric indexes are used when only one of the elements of the statement is blank. A node-centric query is the query that uses the model relative to a particular resource node or edge. The SP2O hash finds outgoing nodes from a resource with a given edge, the PO2S hash finds incoming nodes with a given edge and destination and the SO2P hash finds the edges between two given nodes. According to the project authors, these

---

<sup>2</sup> The user API presented in Redland is a triple-based API one where the triples can be added removed, and searches done by passing in triples with any of the fields allowed to be omitted. This is typically called a *"triples matching" query* and is a common metaphor for retrieval seen in many RDF APIs and applications.

combinations of indexes have been found to be quite useful in experiments, without the need to have full triple hashes combination.

Redland in-memory store uses a dynamic hash configuration so that different stores could set up the hashes they want in any combination. However the in-memory stores are useless when it is necessary to deal with a huge amount of data.

### 3.2. Lucene

Apache Lucene(TM) is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. At the core of Lucene's logical architecture is the idea of a document containing fields of text. This flexibility allows Lucene's API to be independent of the file format. Text from PDFs, HTML, Microsoft Word, and OpenDocument documents, as well as many others can all be indexed as long as their textual information can be extracted.

The indexing process can be divided in three main phases:

- Converting data to text
- Analyzing the text
- Saving the text to index.

Lucene supports two types of indexes, in memory and on disk. Unfortunately the description of these indexes was not found. After analysing other projects that use Lucene we believe that it creates a sophisticated inverted index[9]. It analyses the data and performs additional operations that remove unnecessary parts. For example it removes meaningless tokens, such as stop words (a, an, the etc.) and also the tokens are lowercased and stemmed.

We believe that the used data structures have not been tailored to process RDF data. Lucene may be a good tool for creating sophisticated inverted indexes, but we are sceptic about its capability of indexing big datasets and also offer the required reasoning power.

### 3.3. Jena

Jena [10] [12] is a Java toolkit for manipulating RDF models which has been developed by Hewlett-Packard Labs. The first version (Jena1) used two different database schemas; one for RDBMS and one for BDB. The schema used in RDBMS consisted of three tables:

- Statement – contained all asserted and reified statements and referenced the other two tables. To distinguish the URIs and literals two columns were used.
- Literals – contained all the literals.
- Resources – contained all the resource URIs.

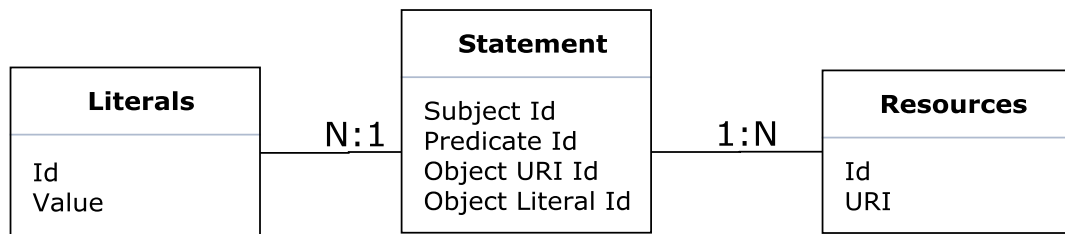


Figure 2 Jena1 RDBMS Schema

The above figure offers the schematic representation of the database scheme. In order to perform a search it was necessary to join the tables, and in some cases it required multiple joins.

Jena1 also offered support for BDB storage. The Jena1 scheme for BDB stored all parts of a statement in a single row and each statement was stored three times, in order to be able to index it by subject, predicate and object.

According to the information that we have found, the BDB storing mechanism was faster than RDBMS. This could be the result of the fact that BDB does not support transactions or to the fact that the BDB schema was denormalized.

The second version (Jena2) implements only the RDBMS schema and trades-off space for time [13]. It uses a denormalized schema by storing encoded URIs and literals in the statement table. The encoding is a necessity in order to be able to differentiate these two types of values. This kind of schema does not require multiple joins when searching but requires a bigger storage space. Nonetheless, in case if the

URI or literal is too big it is saved to separate tables. The schematic representation of the Jena2 database is presented in the following figure [14].

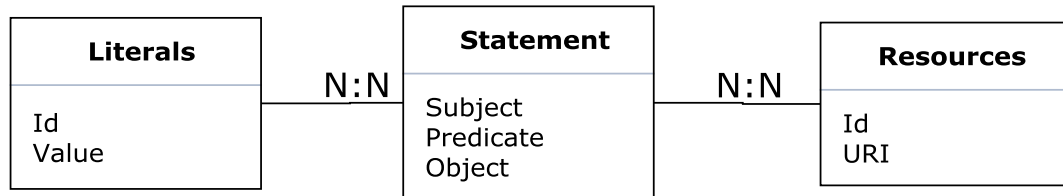


Figure 3 Jena2 RDBMS Schema

In one of the analysed researches [8] it is mentioned that in an attempt to import a large file (hyphen.info<sup>3</sup>) into Jena, using its default RDBMS schema, did not complete even after 24 hours. The preliminary indications were that it was repeatedly refreshing its database indexes during the import.

One of the lately added storage that offers high-performance is a native persistence engine. This engine is called TDB and it uses custom indexing and storage. From the documentation it was not clear who the engine author is, but it is also used in the Fuseki SPARQL server.

A dataset backed by TDB is stored in a single directory in the file system [15]. According to the provided documentation a dataset consists of:

The node table – “The node table stores the representation of RDF terms. It provides two mappings from Node to NodeId and from NodeId to Node. This is sometimes called a dictionary. NodeIds are 8 byte quantities. The Node to NodeId mapping is based on hash of the Node (a 128 bit MD5 hash - the length was found not to major performance factor). The default storage of the node table is a sequential access file for the NodeId to Node mapping and a B+-tree for the Node to NodeId mapping.” [15] The last sentence is a little confusing and we believe it is not fully correct. It does not make sense to use a B+-tree in order to map the Node to NodeId, because it is enough to use the hash function, which will provide the required NodeId. Of course there is a big chance that the documentation does not provide all the details, and the B+-tree is indeed necessary.

---

<sup>3</sup> Hyphen.info is a RDF dataset, which describes computer science research in the UK that is used as a foundation on which to build Semantic Web applications and investigate tools.



Triple and Quad indexes – “Quads are used for named graphs, triples for the default graph. Triples are held as 3-tuples of NodeIds in triple indexes - quads as 4-tuples. Otherwise they are handled in the same manner. The triple table is 3 indexes - there is no distinguished triple table with secondary indexes. Instead, each index has all the information about a triple.” [15] From the above description it is not clear what data structure is used, and how it is used when it is necessary to evaluate a query. By the fact that it creates three indexes, we may assume that each index is a mapping between a triple part (s, p, o) and the rest of the triple. If this is true, then the index will not be very effective when it will be necessary to evaluate a query that contains constraints for two different triple parts. For example, if we know the subject and predicate, we will have to use one of the three indexes, let us say the subject index and then sequentially iterate the result list in order to select the results that contain the required predicate.

The prefixes table – “The prefixes table uses a node table and an index for GPU (Graph->Prefix->URI). It is usually small. It does not take part in query processing. It provides support for Jena's PrefixMappings used mainly for presentation and for serialisation of triples in RDF/XML or Turtle.”

Taking in account that Jena is one of the most frequently used frameworks for building Semantic Web applications, we have decided to give it a try, and to test its performance. Also according to the project change log, a new version has been released in 2011.

### 3.4. YARS

The goal of the YARS project was to create an index that will enable fast retrieval of quads, given any combination of S, P, O, C (context) [6]. This index structure consists of two parts: the lexicon and the quad index. The lexicon stores mappings from literals and URIs to object Ids (OIDs) which are number identifiers used for compactness; the quad index stores the structure information. A considerable influence on the index design had the fact that it was necessary to avoid disk seeks. In order to do it the space was traded for retrieval time, by storing some

information redundantly, in different storing order. The last allows retrieving any access patterns with a single index lookup.

YARS uses B-Trees as indexing structure for disk storage. B-Trees are a well understood data structure and have good properties regarding inserts and deletions. Conceptually, a (key, value) pair is used, where retrieval based on key yields the value using few disk operations.

The lexicon is used for saving the data (nodes) in the form that it was received and for the creation of a numeric key. The key is a 64 bit value and the node is saved in its byte representation. Compared to the Jena project, the nodes are not hashed. Authors stated that the produced key would be bigger than 64 bits. The lexicon is used for searching nodes using the key and for searching the key using the node. Also at this stage it is possible to create an inverted index that will store word occurrences. The last may be useful if it is necessary to provide full text search.

The structure index is used for saving the keys in quads, for example (1, 200, 300, 411). This allows to lookup any combinations of s, p, o, c directly. The keys that are being searched are represented by '?' or by 0 key. There are 16 possible access patterns, and the easiest solution is to create an index for each pattern, but this will take too much disk space. Since some access patterns overlap, such as (s, ?, ?, ?) and (s, p, ?, ?), it is possible to reuse the same index for different access patterns. In the following table are described the 6 indices that are necessary in order to be able to process any query pattern.

Index	Query patterns
SPOC	(?,?,?,?),(s,?,?,?),(s,p,?,?),(s,p,o,?),(s,p,o,c)
CP	(?,?,?,c),(?,p,?,c)
OCS	(?,?,o,?),(?,?,o,c),(s,?,o,c)
POC	(?,p,?,?),(?,p,o,?),(?,p,o,c)
CSP	(s,?,?,c),(s,p,?,c)
OS	(s,?,o,?)

Table 1 YARS list of indexes [6]

In order to create the index on the disk, the keys are concatenated. This technique saves space. Concatenated keys consist of a sequence of OID. The quads are ordered according to the index type. For example in the SPOC index the quad is ordered in the (s, p, o, c) form, and in the POCS index the quad is ordered in the (p, o, c, s) form.

These quads are stored in the corresponding B-tree, with the value part empty. When inserting the key into the B-tree, it is necessary to maintain the order on the first part, then on the second part and so on. As a result the B-tree saves the keys in a sorted order.

With the sorted index it is possible to perform range queries. For example in order to find the entries that have the subject with the OID 1, we just search for all quads in the interval (1, 0, 0, 0) and (1, MAX, MAX, MAX). It is important not to forget to determine which index has to be used; it depends on the query structure.

Also it is possible to add statistics that are useful when trying to find out how many entries there are in a quad interval. In order to do this they insert with each key, four other keys: (0, 0, 0, 0), (s, 0, 0, 0), (s, p, 0, 0) and (s, p, o, 0). In this case the value associated with the quad is the total number of the inserts of the key.

The last version of the project was released in 2006 and uses the BDB for storing the B+-Trees.

### 3.5. 3Store

The goal of the 3Store project has been to design and implement a system for scalable storage of RDF data [16][17]. The back-end storage is provided by MySQL. No other back-end storage has been provided in order to allow tighter integration and higher optimisation. The last known release was in year 2006 and according to AKT [18] it may hold over 30 million RDF triples, but it is a proprietary project.

The arrangement of tables in the SQL database schema is shown in the following table. The used schema is normalised and uses hashes as foreign key. Resources and literals are stored in different tables but use the same hashing function

when the key is created. This is why an additional flag is necessary, which indicates whether the object of the triple is a literal or a resource.

A 64 bit hash was chosen, because it is the largest integer format that is natively supported by commonly used servers. Picking an integer size that can be handled natively and is supported by the database allows for more efficient indexing and therefore joining of tables, which is an important factor in the query performance of the system.

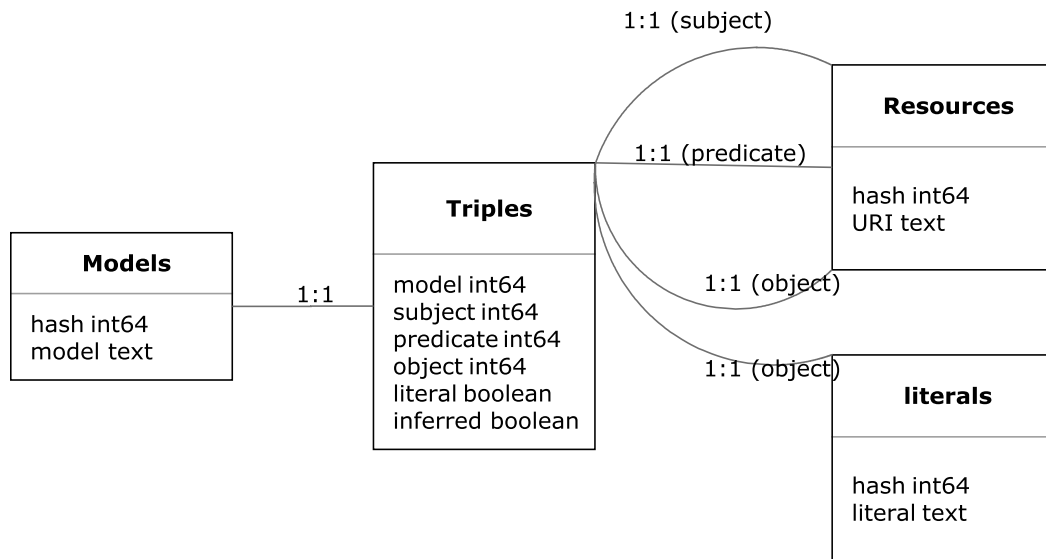


Figure 4 3Store RDBMS Schema

The developers had to choose between creating a hash key using the CRC-64 algorithm or MD5. While MD5 produces a 128 bit output, only 64 are used. The portion of the MD5 hash which is used is unimportant as the blocks are independent and unbiased. After benchmarking<sup>4</sup> these two algorithms the MD5 was chosen to be more fitful.

<sup>4</sup>. The test consisted of hashing 8 random URIs in sequence 100,000 times and recording the number of CPU cycles consumed by each hash function. The test was performed on an Intel Pentium III, the code was generated by gcc 3.2 and the hashing implementations used were taken from GNU libc. On average the CRC-64 function consumed 1062 cycles per hash and MD5 consumed 1091. However the MD5 function is a universal hash (a hash function is universal if it has the property that the probability of a collision between keys x and y when x = y is the same for all x and y), which makes it hard to find colliding keys. It has a large enough output space for this application, unlike more efficient universal hashes, such as UMAC.

### 3.6. Sesame

The last but not the least interesting project that we will analyse is Sesame [19] [20]. “Sesame is being used in industries such as pharmaceutical, healthcare and manufacturing for integrating disparate data sources and as a flexible data storage solution. Sesame is used worldwide by some of the world’s largest companies and government agencies.” [19]

Sesame offers a native triple-store but it also supports RDBMS stores, such as PostgreSQL and MySQL. According to other sources [21] it also supports other triple-stores, including Mulgara, Virtuoso Universal Server and AllegroGraph. The last two are commercial products that according to the description that we found, seem to be very effective, but it was impossible to find any resources that would describe their index structure.

The native store uses on-disk indexes to speed up querying [22]. It uses B-Trees for indexing statements, where the index key consists of four fields: S, P, O and C. The order in which each of these fields is used in the key determines the usability of an index on a specific statement query pattern: searching statements with a specific subject in an index that has the subject as the first field is significantly faster than searching the same statements in an index where the subject field is second or third. In the worst case, the 'wrong' statement pattern will result in a sequential scan over the entire set of statements.

By default, the native repository only uses two indexes, one with a subject-predicate-object-context (SPOC) key pattern and one with a predicate-object-subject-context (POSC) key pattern. However, it is possible to define more or other indexes for the native repository, using the Triple indexes parameter. This can be used to optimize performance for query patterns that occur frequently. These patterns may look similar to those that are used in the YARS project, but they do not allow partial key patterns as OS.

Creating more indexes potentially speeds up querying (a lot), but also adds overhead for maintaining the indexes. Also, every added index takes up additional disk space.

From the above description it is clear that the YARS and Sesame projects have something in common. However, the Sesame project creates fewer indexes and as a result uses lesser space. Taking into account that Sesame is a more frequently used framework, and is currently maintained by a commercial software company, it will be more interesting to test its performance.

### 3.7. Conclusion

We have described some of the existing projects, and tried to explain the used index structures. Many of the described projects do not offer any explicit description of the index structure. Each project is different in a way or another; however, all of them use a RDBMS or a native triplestore. Of course using a RDBMS is not a bad choice, but the experiments that have been conducted by other research groups have proved that the native triple-stores are usually faster. This is probably one of the facts that will have the biggest impact on the design of our RDF index. The big plus of the solutions that use RDBMS is the fact that they are simple. Normally a project that uses a RDBMS defines only the database schema and some internal procedures or functions. When using a RDBMS it is not necessary to bother about ACID properties, everything is out of the box.

We believe that the two most promising projects are Sesame and Jena. These are the projects that are still maintained and are most frequently used. Both frameworks support RDBMS and native triple-stores. Both native stores use custom implementations of B-trees. Another important fact is the offered API documentation. From previous experiences we may say that it is hard to compare projects that are poorly documented. We believe that after implementing our index it will be necessary to implement smaller projects that will use Jena and Sesame libraries. Only in this way we will be able to conduct the necessary tests and compare their performance.

## 4. The design

After analysing related projects we will try to offer our own solution. Projects, which we have mentioned in the previous chapters, usually use: a special hash function, an existing RDBMS or a B-Tree structure. All of the existing projects may be useful and are well designed. Unfortunately, from the existing descriptions we cannot say how they are implemented. We have to limit ourselves to an abstract explanation, which contains the general description of the most important structures or algorithms that are used.

The solutions that use RDBMS are simple and easy to implement, however when it comes to storing big amounts of data they are not as effective as desired. Joining multiple or big tables may be very slow. RDBMS use different index structures but the most frequent are based on variations of B-Tree structure. Using just hash tables is impossible because many of the searches will not contain the full triple. Of course it is possible to create hash tables that will cover all possible combinations, as in the Redland project, but this will generate large amounts of data and eventually will be slow. In our opinion the best solution has to combine the hashing and RDBMS approaches.

### 4.1. Supported Queries

Before trying to offer a new index structure it is necessary to determine what kind of queries it will have to answer. The four main query types that come at first in consideration are insert, update, delete and the most important select. Taking in account the fact that it is necessary to design an index structure that will be used primarily for searching, the update and delete statements will not be analysed. The insert operation will be performed by the core application that will process the triple and will insert it according to the index structure requirements. As a result the only supported way to insert a triple is to process an RDF document.

At this stage we are not trying to define a subset of supported select queries of one of the existing query languages. For the start it is important to understand that the main two select queries will be the full match search and the range search. In the

first case it is expected to receive a full triple and to be able to answer if the triple is in the index. The second case is a little more complicated because only a part of the triple will be known and it will be necessary to return all triples that contain this part. By the term “part” we mean that the subject, predicate or object may be omitted. A simple example of such a query may be the requirement to return all the streets from Prague. In this case the subject will be “Prague”; the predicate may be “is a street” and it is necessary to find all the objects that are represented by street names, which are not specified in the query but match this subject and predicate. A more complicated query would specify other constraints. For example, it may be required to return only the streets that are longer than 200 m. This information can be specified in another triple, in which the street title will be the subject and the length will be the object. In this case a solution would be to find all the streets that are in Prague and then filter them according to their length property. Another solution would be to create two sets. The first would contain all the streets from Prague and the second would contain all the streets that are longer than 200 m. The intersection of these two sets would represent the query solution. A general way in which it is possible to answer to such queries would involve recursively searching the index structure or searching the intersection of multiple sets. Both have their pluses and minuses. In the first case it will be mandatory to traverse the index multiple times. At first it will be required to find all streets, and after it we would search their lengths, this means it will be necessary to traverse the index for each street. In the second case it will be required to traverse the index only once for each set, nevertheless it will be necessary to allocate more space for the result sets and to calculate their intersection. Taking into account that the purpose of this project is to propose a fast index structure, it is desirable to use the second method; however it is also necessary to find out if it is possible to use the first method by minimising the number of traversed entries. This means that it will be clear in which part of the index to search the information about the street length.

It is not reasonable to try and design a new query language; this is why we will just select a subset of the SPARQL query language. This query language has been designed to meet the use case and requirements identified by the RDF Data Access Working Group [23]. SPARQL allows for a query to consist of triple



patterns, conjunctions, disjunctions, and optional patterns [24]. The initial version of the SPARQL language does not support the insert, update nor delete statements, however exists an extension called SPARQL/Update that implements these statements [25]. A detailed grammar specification of the SPARQL query language is not the subject of this paper, but it can be found on the W3C web pages.

#### 4.1.1. SELECT statement

At first we will describe a simple SPARQL query. In order to do so we will use the example that finds the URL of a contributor's blog [26]. The following schema shows the structure for a single contributor and the full model simply repeats this structure for each blog.

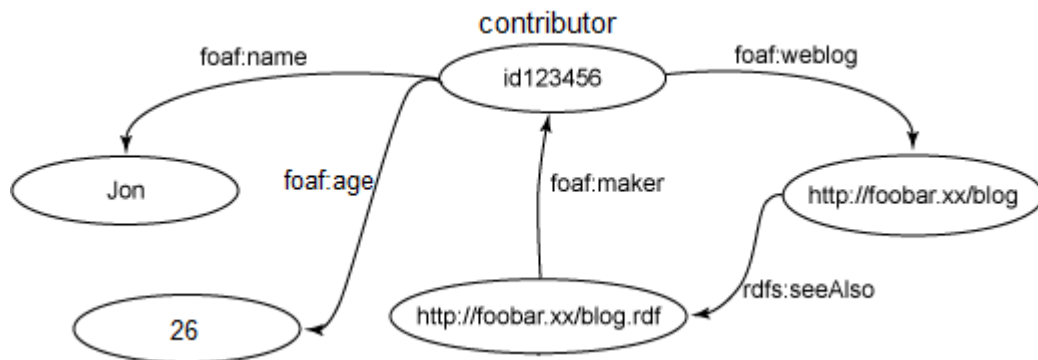


Figure 5 Basic graph structure for a single contributor in bloggers.rdf [26]

The query that performs the required operation is the following:

```

(1)   PREFIX foaf: <http://xmlns.com/foaf/0.1/>
(2)   SELECT ?url
(3)   WHERE {
(4)     ?contributor foaf:name "Jon" .
(5)     ?contributor foaf:weblog ?url .
(6)   }
  
```

The `PREFIX` is used to define an abbreviation so it is not necessary to type the full namespace URI. The `?` or `$` denote that this is a SPARQL variable; both are interchangeable. The `SELECT` clause specifies what the query should return, in this case the `?url` variable. Finally, the `WHERE` clause uses the Turtle-based syntax to specify a series of triple patterns (graph pattern) [27].

In this example, the first triple (subject, predicate, and object) on the fourth row matches a node with a `foaf:name` property of “Jon” and binds it to the variable

named `contributor`. The second triple, on the fifth row, matches the object of the contributor's `foaf:weblog` property and binds it to the `url` variable, forming a query solution (id123456).

The presented select query is straightforward and it is necessary to be able to extend it by adding supplementary conditions. Other useful constructions are optional matches, alternative matches and value constraints. But before explaining other constructions we will present one of the most interesting and complicated select queries. We believe that it is complicated because if translated to a SQL query it would require multiple join operations.

```
(1)    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
(2)    SELECT s1 p1 ?o1
(3)    WHERE {
(4)      s1 p1 ?o1 .
(5)      ?o1 p2 ?o2 .
(6)      ?o2 p3 ?o3 .
(7)      ?o3 p4 o5 .
(8)    }
```

The above query is searching for the triple `<s1;p1;?o1>` that binds to the triple `<?o3;p4;o5>` through other two triples. This may be represented in the following way:

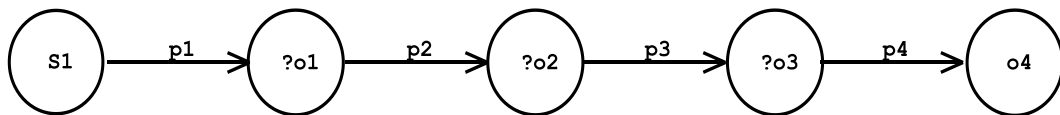


Figure 6 Schematic query evaluation example

#### 4.1.2. Optional match

The optional match is introduced by the `OPTIONAL` keyword. It defines additional graph patterns that bind values when there is a match and do not exclude results that do not match. The following query would return all contributors that are named “Jon” even if they do not have any match for the clause specified on the sixth row.

```
(1)    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
(2)    SELECT ?contributor ?url
(3)    WHERE {
(4)      ?contributor foaf:name "Jon" .
(5)      OPTIONAL {
(6)        ?contributor foaf:weblog ?url .
(7)      }
(8)    }
```

#### 4.1.3. Union

The alternative match is used when it is necessary to return whichever of the properties is available. In order to be able to perform this kind of query the graph patterns have to be connected using the `UNION` keyword. The following query will select all contributors that are named “Jon” and have at least one match for the clauses specified on the sixth row. This means that each “Jon” has to have bidding for the variable `?url`, which matches the clauses that have the subject `?contributor` and the predicates `foaf:weblog` or `foaf:maker`.

```
(1)    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
(2)    SELECT ?contributor?url
(3)    WHERE {
(4)      ?contributor foaf:name "Jon" .
(5)      {
(6)          {?contributor foaf:weblog ?url}
              UNION
              {?contributor foaf:maker ?url}
(7)      }
(8)    }
```

#### 4.1.4. Filter

The value constraint is introduced by the `FILTER` keyword. This is used when it is necessary to restrict the values of the bound variables. The value constraints are logical expressions that evaluate to Boolean values, and may be combined with logical `&&` and `||` operators. The following query will select all contributors that are named “Jon” and have the binding value of the variable `?age`, specified in the pattern on the sixth row, bigger than 20 and smaller than 30.

```
(1)    PREFIX foaf: http://xmlns.com/foaf/0.1/
(2)    PREFIX xsd: http://www.w3.org/2001/XMLSchema#
(3)    SELECT ?contributor?url
(4)    WHERE {
(5)      ?contributor foaf:name "Jon" .
(6)      ?contributor foaf:age ?age
(7)      FILTER xsd:int(?age)>20 && xsd:int(?age)<30
(8)    }
```

### 4.2. Index design

Now that we know how the queries will look like, we will try to analyse a few data structures that we believe may fit our requirements. We will try to describe the general characteristics and after that it will be necessary to decide if it makes any

sense to make a more detailed analyse. As we have mentioned earlier, we would like to divide the indexation into two parts. The first part would be the hashing of the RDF triple. The second would be the insertion of the hashed value into an appropriate indexation data structure.

#### 4.2.1. Hashing

At first it is necessary to prepare the triples. As it has been described in the analysed projects, saving the triple in the initial form in an index is too expensive. A good solution to this problem is to clean the data and to create a hash. Creating a hash is not hard and there are multiple hash algorithms that can be used, but in order to be able to answer to queries that will contain only a part of the triple (subject, predicate or object) it is necessary to create a hash for each part. The length of the hash may be different and for this project it is necessary to select the smallest possible. The size depends on the selected hashing algorithm but it is also necessary to take into account that a universal hash function is the most appropriate. It is not the purpose of this work to create a universal hash function that would create the smallest possible hash, that's why an existing algorithm will be used. The known hash algorithms usually produce at least 128 bit hashes, while the most appropriate size would be 64 bits. The size of the appropriate hash is determined by the necessity to minimise storing space and the best compatibility with the hardware. The 64 bit integer is the largest integer format that is natively supported by existing servers. We did not conduct our own benchmarking, but according to the tests that have been performed by the team, which developed the 3Store project, the best algorithm is MD5. The hash blocks that are produced by this algorithm are independent and unbiased making it possible to trim the hash to the required length. It has a large output space and in the context of our datasets it is hard to find colliding keys. As a result we have also decided to use the MD5 hash algorithm as one of the most effective and easy to use.

Now the question is what part of the triple has to be hashed? In some of the analysed projects that did also used some kind of hashing, the indexes contained the hashed value of all or just some of the possible triple parts combinations. For

example in the Redland project there are four types of hashes, while YARS project uses six types of combinations.

We may not currently decide how many hashes it will be necessary to create. It all depends on the data structure that we will be using, but we will need a mechanism that will permit us to obtain the textual representation of the triple only by knowing the hashes.

#### 4.2.2. Full text search

At this stage it is expected that the processed query contains the subject, predicate and object in the same form that it has been specified during indexation. This means that if during indexation the object of the triple X was “Jon Mathew”, the query that contains in the condition clause the value “Jon”, would not return the X triple. A full text search would definitely be an interesting feature; however it will complicate the index structure. We believe that this feature may be added later if it will be required.

#### 4.2.3. Ontologies as a dimension

An interesting question is the possibility to use ontologies formulated in RDFS or OWL as one of the dimensions. RDF data may use different vocabularies and even one document may contain definitions from multiple vocabularies. This means that two different definitions may have the same meaning and it is hard to say if it is possible to identify such cases. Ontologies are more like social contracts. The answer to this question may also be deducted from the affirmation made in the YARS project description. The developers of the YARS index have stated that the use of ontologies as a schema for relational databases will eliminate one of the important advantages of RDF compared to other storage mechanisms – the ability to simply merge different datasets without the necessity to merge the ontologies first [6].

#### 4.2.4. Full triple index (FullDoc)

Before proceeding further we have to notice that currently we are dealing with two main types of objects. The first is the full triple text representation, that may

have variable size, and the second is the hashed triple that is composed from three 64 bit values. The idea is to be able to evaluate the select query using the second type of objects. The reason is simple; the second type of objects will occupy lesser space. This means that it will require lesser disk reading operations and manipulating with numbers is easier than with text. After evaluating the select query it will be necessary to bind the set of hashes that represent the result, to the correct full triple text representations. The best way to do it is to use an inverted index. This means that the triple will be uniquely identified by a key (FullDoc key), and the hashed triple will contain this key. As a result the hashed triple will contain one more piece of information, the FullDoc key.

The FullDoc will probably be the easiest index to build, because it will be constructed as an inverted index. In this case we will use a very good understood data structure, the B+-tree<sup>5</sup>. Inserting and finding a record requires  $O(\log_b N)$  operations, where  $b$  is the number of items in a node and  $N$  is the total number of items in the tree. This data structure is very attractive to us, because in addition to efficient retrieval the B+-tree can be kept balanced at a small cost during insertions. Leaves will contain the full triple text, while the internal nodes will contain the keys. We got two alternatives in which way to build the key. The first and the easiest would be to provide a unique number to each triple. The second is to use once again the hash function and to build a hash for the full text representation. We have decided that the second option is the one that makes more sense.

The first solution does not take in account the fact that the same triple text representation may occur more than once. As a result by indexing the same data the FullDoc will not merge the similar items. The second solution is much more effective in this context. Similar triples will raise a hash collision, which means that we will just have to compare the two triples text representation.

---

<sup>5</sup> The principal advantage of B+-trees over B trees is they allow you to in pack more pointers to other nodes by removing pointers to data, thus increasing the fanout and potentially decreasing the depth of the tree. The disadvantage is that there are no early outs when you might have found a match in an internal node. But since both data structures have huge fanouts, the vast majority of your matches will be on leaf nodes anyway, making on average the B+ tree more efficient.

An additional plus of the second solution, is the fact that we will be able to say if the triple is contained in the index, without accessing other indexes. It is not clear if this functionality will be used in a way or another, but it is good to know what functionality it may offer.

A more detailed description of the FullDoc index implementation will be provided in future chapters.

#### 4.2.5. Data structures

The index structure will contain elements composed from four 64 bit integers and it is clear that it is necessary to operate with multidimensional data formed from 4 dimensions. Three of the 64 bit integers will represent the hash of each part of the triple and the fourth will be the FullDoc key. From the previous description it is obvious that the three hashes have to identify the FullDoc key, which means that it does not have to be a part of the composed key as it will be the determined value. This simplifies the problem; however, we still have to operate with multidimensional data composed from 3 dimensions.

Simple B-trees are not appropriate when it comes to working with such elements. B-trees are fast and are very good understood but for indexing multidimensional data, it is necessary to use a different structure. One of the options is to use the R-tree structure. This is one of the preferred methods when it comes to indexing multidimensional data. Another data structure that may be analysed is the multidimensional B-tree. By understanding this data structures it will be possible to design an index that will fit the requirements.

Also an option would be to reduce the multidimensional data to one dimension. This may be done in a simple way, just by creating an index for each dimension. For example, the key may be the hash of the subject and the value may be the remaining two hash values plus the FullDoc key. In this case we could use the B+-tree as the main data structure. The plus of this approach is the simplicity and effectiveness that the B+-tree may offer. Such an index can easily evaluate a query that specifies only one select pattern; however, in order to evaluate a query with multiple patterns it may be necessary to traverse the B+-tree multiple times.

Before making the final decision it is necessary to analyse the effectiveness of the selected data structure in accordance to the select query requirements, which were described earlier. It is important to understand if the data structure can be used to answer to the specified queries and how intensive will be the use of the secondary memory while fetching the results.

#### 4.2.6. R-tree

In order to handle multidimensional data efficiently, a database mechanism needs an index mechanism that will help retrieve data items quickly according to their dimensions and values. An R-tree is a height-balanced tree similar to a B-tree with index records in its leaf nodes containing pointers to data objects [28]. Nodes correspond to disk pages and the structure is designed so that the search operation requires reading only a small number of disk pages. The R-tree is completely dynamic and no periodic reorganization is required [29].

The R-tree contains tuples that represent spatial objects, and each tuple has a unique identifier which can be used to retrieve it. A tuple is defined as **(I, tuple-identifier)**, where **I** refers to the smallest binding n-dimensional region (MBR) that encompasses the spatial data pointed to by its tuple-identifier. **I** is a series of closed-intervals that makeup each dimension of the binding region.

**Example:** in 2D,  $I = (I_x, I_y)$ , where  $I_x = [x_a, x_b]$ , and  $I_y = [y_a, y_b]$ .

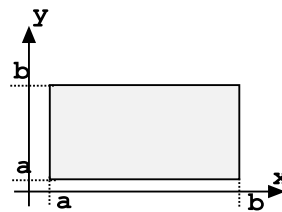


Figure 7 R-tree binding region

In general  $I = (I_0, I_1, \dots, I_{n-1})$  for n-dimensions, and  $I_k = [k_a, k_b]$ . If either  $k_a$  or  $k_b$  (or both) are equal to  $\infty$ , this means that the spatial object extends outward indefinitely along that dimension.



A tuple  $E$  in a non-leaf node is defined as  $E = (I, \text{child-pointer})$ , where the child-pointer points to the child of this node, and  $I$  is the MBR that encompasses all the regions in the child-node's pointer's entries.

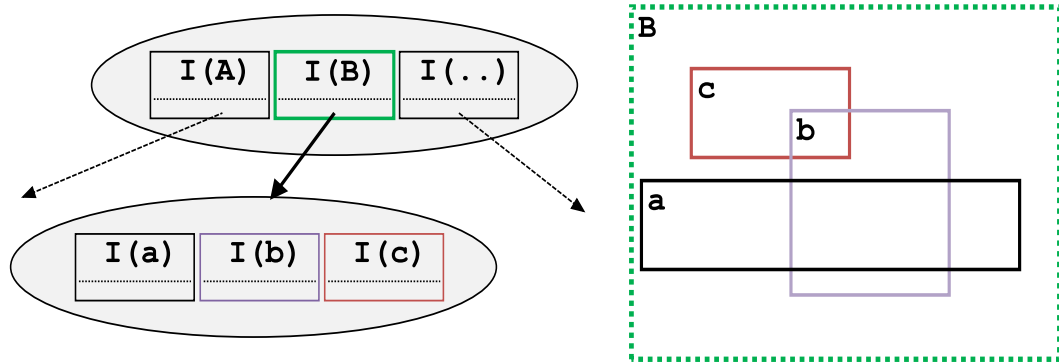


Figure 8 R-tree structure example

Let  $M$  be the maximum number of entries that will fit in one node. Let  $m \leq \frac{M}{2}$  be a parameter specifying the minimum number of entries in one node. Then an R-tree must satisfy the following properties:

1. Every leaf node contains between  $m$  and  $M$  index records, unless it is the root.
2. For each index-record Entry  $(I, \text{tuple-identifier})$  in a leaf node,  $I$  is the MBR that spatially contains the  $n$ -dimensional data object represented by the tuple-identifier.
3. Every non-leaf node has between  $m$  and  $M$  children, unless it is the root.
4. For each Entry  $(I, \text{child-pointer})$  in a non-leaf node,  $I$  is the MBR that spatially contains the regions in the child node.
5. The root has two children unless it is a leaf.
6. All leaves appear on the same level.

A **Node-Overflow** happens when a new Entry is added to a fully packed node, causing the resulting number of entries in the node to exceed the upper-bound  $M$ . The ‘overflow’ node must be split, and all its current entries, as well as the new one, consolidated for *local* optimum arrangement.

A **Node-Underflow** happens when one or more Entries are removed from a node, causing the remaining number of entries in that node to fall below the lower-bound  $m$ . The underflow node must be condensed, and its entries dispersed for *global* optimum arrangement.

As any other structures, R-tree has its minuses. One of the known minuses is the minimum fill of 30%-40% of the maximum number of entries, while B-trees guarantee 50% page fill. The reason for this is the complex balancing required for

spatial data as opposed to linear data stored in B-trees. Also it has been noticed that the R-tree is more effective when it is used for object indexation (special object) and is lesser effective when it is necessary to index points.

The height of an R-tree containing  $N$  records is at most  $\lceil \log_m N \rceil - 1$  where  $m$  is the minimum number of items in a node and  $m \leq \frac{M}{2}$  where  $M$  is the maximum number of items in a node.

#### **4.2.6.1. Searching**

The search algorithm descends the tree from the root in a manner similar to a B-tree. However, more than one sub-tree under a visited node may need to be searched; therefore it is not possible to guarantee good worst-case performance. Even so with most kinds of data the update algorithms will maintain the tree in a form that allows the search algorithm to eliminate irrelevant regions of the indexed space, and examine only data near the search area. If compared to a B-tree, the R-tree does not offer any performance guarantees, but according to some sources, generally performs well with real-world data.

1. Given an R-tree whose root node is  $T$ , find all index records whose rectangles overlap a search rectangle  $S$ .
2. If  $T$  is not a leaf, check each entry  $E$  to determine whether  $E$  overlaps  $S$ . For all overlapping entries, invoke Search on the tree whose root node is pointed to by child-identifier.
3. If  $T$  is a leaf, check all entries  $E$  to determine whether  $E$  overlaps  $S$ . If so,  $E$  is a qualifying record.

#### **4.2.6.2. Effectiveness**

Now that the structure of the R-tree has been explained and we have to emphasize that each dimension will be represented by one of the three hashed parts of the triple. In the case of the simplest query (the full match query) it is necessary to find the node that matches the specified values. If such a node cannot be found, this means that it does not exist.

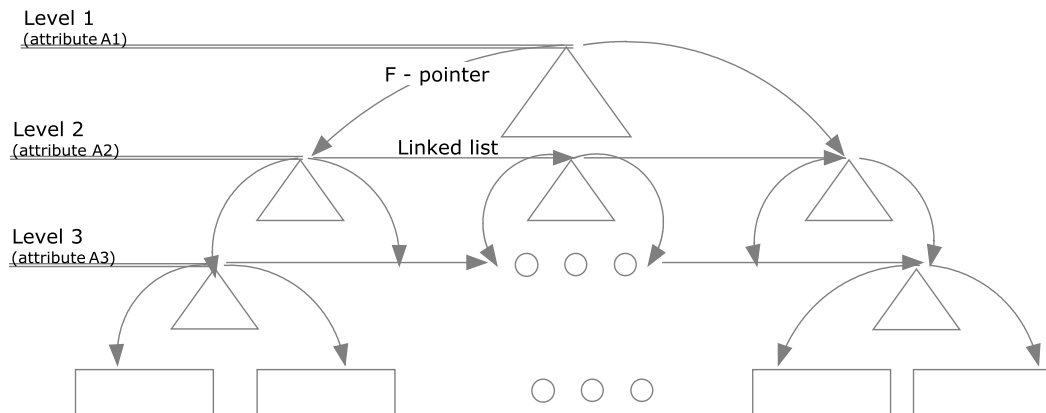
The second type of query, the one that specifies just a part of the triple, also can be answered using the R-tree. The missing parts will be replaced by the minimum and maximum hash values. This means that all the triples that are in the R-tree, which match the specified values and have the value of the unspecified dimension in the interval of the minimum and maximum hash values are query results.

A more interesting case is the situation when it is required to answer to a query that contains multiple triple patterns. This is the case when it is necessary to find all the streets in Prague that are longer than 200 m. Earlier it was noted that it is desirable to minimise the number of index traversals and this could be achieved by knowing in which part of the index to search the required information. Unfortunately in this case the R-tree structure does not provide such information. Even if the triples have something in common, in our case the street title, in different select patterns, the searched value may be the subject or the object. This means that it is necessary to change the dimension according to which the search is performed, and in order to return all matching results it is necessary to start a new search.

#### 4.2.7. Multidimensional B-trees (MDBT)

Multidimensional B-tree is very similar to the B-trees and, as the name implies, it uses B-trees to maintain each dimension in a manner that will guarantee efficient retrieval and low cost maintenance [30] [31].

The schematic MDBT organization is presented in the following figure.



**Figure 9** Schematic representation of a multidimensional B-Tree

Each tree level corresponds to a different attribute, where the nodes represented by triangles are themselves B-trees (sub B-trees). Sub B-trees may have a different order and size. The structure of a node in the B-tree for the attribute (key) **A<sub>i</sub>** is:

$$(K_1 P_1 F_1) \dots (K_m P_m F_m)$$

where **K<sub>j</sub>** is the **j**-th value of the attribute **A<sub>i</sub>**; **P<sub>j</sub>** points to a node at the next level in the same B-tree containing values of **A<sub>i</sub>** between **K<sub>j</sub>** and **K<sub>j+1</sub>**; and **F<sub>j</sub>** point to a B-tree at level **i+1**, which contains the set of values of attribute **A<sub>i+1</sub>**, which appear together with **K<sub>j</sub>**. A more detailed description of the MDBT may be found in the provided sources.

Another important feature is the presence of the ordered linked lists at each level. These linked lists are used in the case of partial match search, which will be described later.

#### 4.2.7.1. Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. The difference is that when the value of one attribute at level **i** is located, it is necessary to follow the link to the next B-tree at level **i+1**.

In case of a search with missing attributes it is necessary to use the ordered linked list from the level of the known attributes. The search strategy for partial match query is represented in the following figure.

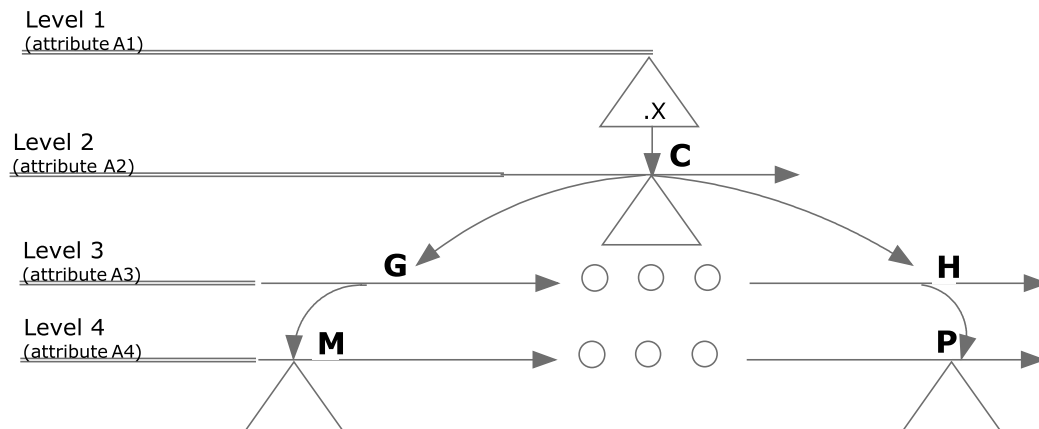


Figure 10 MDBT Partial match example

The above figure corresponds to a query, which specifies the attributes only for the first level. The search at the first level yields the value **x** of the attribute **A1** and its corresponding pointer to the B-tree from the second level with root **C**. Since the value of the attribute **A2** is not specified, it is necessary to retrieve the pointers to the left and right sub B-trees, **G** and **H**. These two pointers delimit the portion of the linked list from the third level whose parents reside in the B-tree designated by **C**. Since the attribute **A3** is not specified either, it is sufficient to access only roots of the B-trees denoted by the **G** and **H** and to proceed to the next fourth level. The previous operation returns the pointers **M** and **P**, which delimit the portion of the accession list containing the relevant record addresses.

According to the sources that describe the MDBT, in the worst case scenario the retrieval time is of complexity  $O(N)$  where **N** is the number of items. A more detailed description of the MDBT can be found in source [30].

#### **4.2.7.2. Effectiveness**

From the above description it is easy to denote that each of the three hashed parts of the triple will represent an attribute and the data page will contain the FullDoc key. In the case of the full match query it is necessary to traverse the tree as described earlier. The partial match query has been described earlier.

Once again a more interesting case is the situation when it is required to answer to a query that contains multiple select patterns. After finding the solutions for one pattern, that will involve the usage of the partial match algorithm, it will be required to evaluate other constraints. This means that it is necessary to traverse the MDBT once again. However, the result set will contain only the FullDoc key, which cannot be effectively used while traversing the MDBT the second time. This means that we will have to fetch the triple from the FullDoc index and transform it to a valid MDBT attributes. We believe that such an approach is too expensive.

### **4.3. Conclusion**

We have analysed the most frequently used data structures, which are used when it is necessary to work with multidimensional data, and it is clear that none of

them satisfies all the requirements. The biggest problem, as it was initially expected, is the incapability to answer in an effective way, to a select query that contains multiple patterns. Another minus is the complexness of all analysed data structures and the poor behaviour in the worst case scenarios. The R-tree is usually used with geographical coordinates or geometrical figures. It requires the computation of the area occupied by a multidimensional binding region, which may be a problem in our case. If compared to the multidimensional B-tree, the last is much more appropriate and maybe even more efficient. The problem of the MDBT is the complicated structure and what is more important we currently do not know for sure how we may serialise such data structure. We have to take into account the fact that it will be very big, and all the internal references will be probably transformed to a representation that can be serialized.

After analysing all cons and pros, we have decided to implement three indexes, one for each part of the triple. The indexes will be based on a modification of a B+-tree. If compared to earlier described data structures, this will be easier to serialise and the queries with multiple patterns are evaluated in the same manner. A big plus of the B+-tree is its retrieval time. By creating three indexes, we will simplify the query pattern evaluation. At first it will be necessary to choose, which index to use. This will be done just by analysing, which parts of the triple are not missing, and select the index that corresponds to any of the not missing part. The only problem may be the structure of the leaves that may contain a high number of items. According to the number of items in each leaf, it will be mandatory to decide what is more effective, a linked list or an internal B+-tree (sub-tree). For example, we believe that the predicate index (the key is the predicate hash) will contain only a few nodes in the main B+-tree, but the list of corresponding values will be too big in order to be effectively saved in a linked list. Of course in such case we could use a simple linked list for the predicate keys that will contain pointers to the appropriate B+-tree, but this is not very effective in the moment when we are not able to analyze the structure of the indexed data. For some datasets the above affirmation may be wrong, as a result our index will be slow. By offering the possibility to define the index structure for each field individually we may improve the index effectiveness.

## 5. RDF index design

In this chapter we will explain in detail the design and important implementation characteristics of our RDF index. At first we would like to say that the program is developed in C# and is running on the .Net 3.5 platform. We understand that this fact may be a limitation on portability but the number of similar projects that have been designed to run on the .Net platform is very small, while the number of PCs that use it is very big.

The index is formed from the following four components: full triple index (FullDoc index), subject index, object index and the predicate index. The subject, object and predicate indexes are used for answering to SPARQL queries, while the FullDoc index is used for returning a human understandable result, the textual representation of the triple.

### 5.1. File structure

As any other index or database we had to design a file structure that would permit us to serialise the index in an effective and fast way. We had to take into account the size and the general structure of our index. First of all the fact that the index is formed from multiple components leads us to the idea that each index component has to be saved in separate groups of files. These groups are composed from two types of files: the data files and the structure files. Each group may contain multiple data files, of a limited maximum size, and only one structure file. Initially we were thinking of creating multiple types of files that would have blocks of different size. After a few tests it was clear that such a solution would not be effective. Writing blocks of a different size than the one that is supported by the disk may be slow or may require to access the same disk block multiple times. As a result the data files are divided into 4KB blocks. The structure file (\*.str) contains additional information about each file in the group, such as: the file id, file name, the number of free blocks and the block id of an empty block. In further chapters we will explain the meaning of each of these items.

Initially we wanted to be able to create a distributed index and have allowed the data structure to contain the full data file path. Latter we have understood that it will be very hard to copy the files and to change their location. We have decided that it will be better to use relative paths as a result all index files have to be in the same directory. Our implementation gives the possibility to specify the path to the directory that contains the index files.

## 5.2. FullDoc index

As it has been mentioned earlier the FullDoc index is used for mapping the results of the SPARQL queries that are applied on the other three components. This index contains all indexed triples and there hash code, represented by a 64 bit integer. The used data structure is a B+-tree that uses the hash of the full triple as the key and the triples as values that are saved in the leaves. The triples are contained in the tree only once, this means that each key is mapped to one triple.

This component is one of the easiest to build, but in the same time it is one of the biggest. Because of this component it was necessary to divide the triples into multiple file blocks of 4 KB or in other cases to use a 4KB block that would contain multiple triples. The fact that the size of these items is variable brought many complications, but in the same time made the design process more challenging. Latter we will describe how data is being serialised to a disk block.

## 5.3. Subject and object index

The subject index contains just hashed values. The used data structure is the B+-tree, which uses the subject hash as the key. The predicate, object and FullDoc hash are used as the contents of the leaf items. At first sight this component is very similar to the FullDoc index; however, the main difference is in the leaf item structure. The same subject may be contained in multiple triples as a result the leaf item has to be updated each time the same subject is inserted. The most obvious solution is to use a list. This should be effective enough if the number of occurrences is not too big (1-10). It is very important not to forget the fact that before updating it is also necessary to check if the value is not already contained in the list. The update operation may be very ineffective for a higher number of occurrences. In some cases



it will be required to read all the data from disk and also to iterate through them in order to check if the inserted value is not already contained.

The same as the subject index, the object index contains just hashed values and the used data structure is the B+-tree, but tests have shown, that compared to the subject index, the occurrence of the same object in a document is much more frequent. As a result, the leaves may contain a high amount of items, and the index will not be very effective. This is why it was decided to use internal B+-trees. The internal B+ trees will have the same structure as the tree from the subject index, but leaves will not contain the object hash value. This structure has something in common with the multidimensional B-tree; however, it does not have the linked list that would interconnect the internal B+-trees. In our case the partial matching is done in a much simpler way, just by searching in the right index component. Compared to the subject index we have to be careful with the nodes that may contain just a few items. A low number of repeated internal keys may lead to wasted space not only disk but also RAM, which has to be allocated for the root node.

Another interesting option would be to combine the list and the internal B+-tree solutions and to create an adaptable leaf. In case of a small number of similar key insertions, leaves could contain a list of values, while for a higher number of collisions it will be changed to a B+-tree.

#### 5.4. Predicate index

The predicate index is similar to the object index, however it has to be noticed that experiments have shown that the number of predicates is in many cases a lot smaller than the number of objects. This is why it was decided that the index will be represented by a B+-tree, which will contain in the leaves references to internal B+-trees. We could also use the same idea as in the subject and object indexes; however, the adaptable leaf type has a much more complicated structure and may be slower.

## 5.5. Addresses and data blocks

The first major problem that had to be solved was the development of an effective and easy way of accessing a file block. In order to be able to solve this, we decided that each block has to have an unchangeable internal address. The difference from an in memory index is the fact that internal references are lost after stopping the program. This means that even if we had the data saved into a file, after a program restart we would have to initialise the whole index once again. This would not be a problem if the index would fit in a small file, but we are expecting it to deal with files bigger than 2 GB and the program will work on desktop PCs, which if compared to a server, may offer a much smaller amount of RAM.

As it has already been said the data files are divided into fixed size blocks. Each block in a group of files may be addressed by the file id and the block id (**simple address**). Internally the address is represented by 16 bit (file id) and 16 bit (block id) unsigned integers. As a result the maximum number of files is  $2^{16}$  and the maximum number of blocks in a file is  $2^{16}$ , which means that a file may be of a maximum  $256^6$  MB and the total size of a file group will probably not fit on a normal disk. Of course we may change the size of each of the address parts; as a result it is possible to save some space for other useful data or we can increase the maximum index size.

The simple address is used to address simple blocks. The structure of a simple block is represented in the following figure.



Figure 11 Simple block structure

The simple block occupies 4 KB and is composed from three parts. The first byte represents the block type. At this stage we will just say that the block may be of one of the following types:

- Final

---

<sup>6</sup> 4 KB (block size) \*  $2^{16}$  (nr of blocks)

- Not final
- Smart block

Next, the block contains the byte representation of the useful data. In the end it may contain the address of the next block. Each class that has to be serialized implements two methods that are responsible for transforming the object to a byte array and initialise the object from a byte array. As a result it is possible to replace some classes without affecting the functionality of other classes. This is very useful when we want to change the type of the leaf items.

Some of the saved objects may be bigger than a 4KB block; however, the data object should not contain the necessary logic to deal with this situation. The implementation offers a special layer, which is responsible for dividing such data objects into multiple blocks. This is one of the reasons why it was necessary to introduce the block type and the next address parts. The block type is used to describe whether the block contains (is not final) or not (is final) the address (the next address) of the next block that contains further object data.

Not all data objects that are saved in a file are big enough to occupy a block; this is why it is necessary to be able to save multiple data objects in a single block (I/O costs). Another type of used address is the so called **smart address**, which contains an additional 8 bit unsigned integer that is used for addressing inside the block that will be called the index, in this context. The index is the internal id that has to differentiate the block contents. By setting the index value to zero the smart address may be used as a simple address. At a lower level the smart block address is transformed to a simple block address. Just like the simple address the smart address corresponds to a special type of block that we call the smart block. Compared to a simple block, the smart block has a more complicated structure, which is presented in the following figure.



Figure 12 Smart block structure

In order to be able to modify (delete, insert or update) the contents of a smart block, without the necessity to update any other blocks or items (that could be very

expensive), it was decided that the index has to be mapped, inside the block, to the right offset. As a result it is possible to change the way data are arranged inside the block and in the same time to offer a fixed (unchangeable) external address. This information is saved in the items information part of the smart address, which is composed from multiple items, and each of them describes one data object. The following figure contains the schematic representation of one information item. It contains the index, an offset and the useful data size in bytes. The offset defines in bytes the starting position of the data object and by knowing the size it is possible to calculate the ending position. By knowing the object type, we may call the appropriate method, which will use the byte array to fill with data an instance of the same type.



**Figure 13 Single smart block item information**

The smart block is a part of the simple block as a result it is possible to use all the properties of a simple block. After the creation of the smart block, the block type had to be enriched by a new possible value that differentiates the smart block from the simple block. Two important restrictions are the fact that smart blocks may not contain data objects that have to be saved in multiple blocks and the maximum number of saved data objects is 255. The last is determined by the fact that the index is a byte, which can hold at most 256 values, one of which is used to signalise that the index has to be ignored and the block has to be processed as a simple block.

One of the problems that had to be solved during the smart block implementation was to locate an index that has not been used earlier. At first we wanted to use a simple bit array, however the idea of iterating through the array did not seem too effective. As a result we have decided to use four 64 bit unsigned numbers that would contain the information about the used indexes. Each bit can offer information about one of the indexes. By negating each number we can easily find the first unused index. After a few hours of testing we came to the conclusion that the precision of some of the operations for the 64 bit unsigned integers is not as good as we would expect, as a result we had to use eight 32 bit unsigned integers.

In order to make the I/O operations more effective it was decided to add an additional byte to the address that would signalise if the block is final or it contains the address of the next block. As a result it is not necessary to retrieve the block content in order to see if the block is final or not.

Now that we have explained the addressing system we may calculate the full address size which will require: 2 bytes for the file id, 2 bytes for the block id, 1 byte for the internal smart block id and 1 byte for the block type flag. It is important to understand that we are dealing with simple classes, but in the same time the number of instances that will be created is very big. As a result it is necessary to minimise the number of attributes that each such instance will have.

## 5.6. B+-tree structure

The index structures that will be described further use the B+-tree as the main data structure. The way a B+-tree works is easy to understand and will not be explained, however, the technique in which such a structure may be serialised in an effective way is not as clear and easy as it could look at first sight.

Before beginning the development process we have tried to search for a library that could make our task easier; however, at that moment we could not find any library that would deal with B+-trees that have to be persisted. After nearly finishing the development, we have found a recently released library called BPlusTree developed for .Net 4.0 [32]. The library description has inspired us to try it, but after a test run, we came to the conclusion that it does not fit our needs. At first we shall say that our implementation has a similar structure, which made it possible to use our classes in the same manner as it was required by the library. Nonetheless, we were not able to create a B+-tree for our standard dataset. The library was accessing the same items (leaves) multiple times and was not able to process bigger items in a correct manner. As a result we have decided to complete the development of our own B+-tree implementation.

The next figure represents the internal representation of a node from our B+-tree. It contains three types of information: the keys, the addresses of the child nodes or leaves and additional node information, such as node type and number of keys.

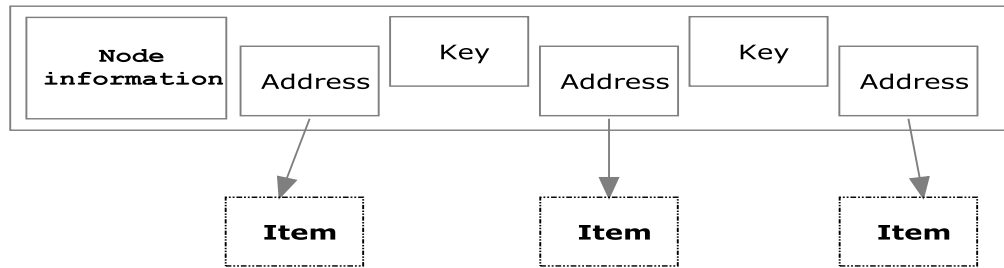


Figure 14 B+-tree node structure

The node also contains information about the node state, which is useful when it is necessary to serialise the node. According to this information it is possible to decide if the node has to be serialised or it has not been changed from the last serialisation. This piece of information is very important and sensitive because it has to be changed each time the node is modified, otherwise there is a chance that the serialised version will not be consistent. Many times we had to deal with the fact that the modification of the child item was not correctly interpreted by the parent node. As a result the address from the parent node was pointing to a wrong data block or the node was serialized even when no modification has occurred.

In order to optimise the I/O operations it was decided that the best tree node size would be the one that would use at most 4KB (disk block size). This means that with items of the size of 8 bytes, the node may contain at most 512 ( $4 \cdot 1024 / 8$ ) items. However, this would be possible only if there would not be any other necessary information, such as the block address. As a result each key requires another 6 bytes for the child item (node) address and one address for the child that contains the keys that are bigger than any of the parent key. This means that the current maximum number of items in a node is 292. Furthermore it is necessary to take into account that we also have to allocate: 2 bytes for the number of items that the node contains; 1 byte for the node type (internal or leaf), and 1 byte that the block itself requires for internal usage (block type). In conclusion the node may contain at most 291 items.

In case of internal B+-trees it will be required to create nodes with fewer items. By doing so, we will be able to minimise the wasted disk space. This type of nodes will be written into smart blocks, and in order not to be forced to update the smart block structure at each insertion, it will be wise to allocate the required space

at the beginning or to use a lazy serialization technique. The last means that the item is saved only when it has to be removed from the RAM.

## 5.7. Empty blocks

One of the problems that is necessary to solve, when working with a file that is divided into blocks, is the block management. First of all we have to know which block is empty and may be used. Then it is necessary to be capable to reuse blocks that have been once used and later emptied. In order to solve this problem we have decided to create a list of empty blocks that will be available even after restarting the program. The idea is very simple, we have the address of an empty block, which is saved in the structure file, and the empty block with the earlier mentioned address contains the address of the next empty block. In such way, by following the addresses contained in these blocks, we may iterate through the list of empty blocks. A new empty block may be added by inserting it in the head of the list. The following figure displays the way the block C is emptied and added to the head of the empty blocks list, pointing to the block that was initially in the head of the empty blocks list. The empty blocks are denoted by the letter X.

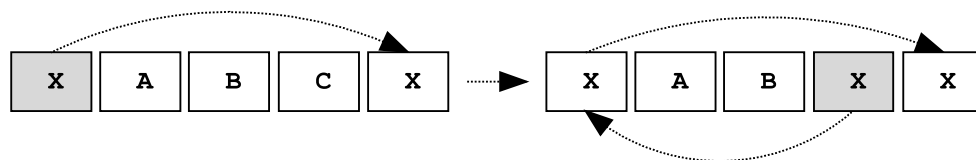


Figure 15 Adding a new empty block

The last empty block will not contain any address and this means that all the blocks contained in the file, which have a higher address, are empty. For example, before any data are saved to the index, the structure file does not point to any empty block and this means that the file is empty and we just increment the block id.

Another option would be to save multiple addresses in the same block; as a result the number of read operations would be smaller. However, we decided not to implement this improvement, because during the insertion process, the number of deleted blocks is not very big. Normally we just allocate blocks that have never been used before and their address is calculated according to the address of the empty block that is saved in the structure file.

## 5.8. Serialization effectiveness

When speaking about effectiveness it is clear that the most expensive operations are the I/O operations, this is why nearly all optimisations are oriented to decrease the number of I/O operations.

The optimisations that we came up with are the following:

- Write larger sequences of blocks – this may be effective, but the blocks are accessed in a random manner and it is impossible to order the blocks or at least this would be very complicated. In conclusion, this method will not be implemented. On the other hand the operating system may be using its own I/O buffer.
- Use a cache– this is the most effective optimisation. It may be implemented at different levels, for example, at the item or at the file block level. The possible types of cache will be described in the next chapter.
- Differentiate the types of data - as it has been said earlier, data blocks are written into multiple files of the same maximum size. In order to make the process more effective we have decided to divide the files according to the content. For example, this could mean that the data from the leaf items are not written to the same files as the data from internal nodes. This is very important in cases when the internal nodes and leaf items may use the same block type. Mixing different types of information may lead to the need to fetch more disk blocks. It is clear that the internal nodes are accessed with a higher frequency, and it would be ineffective to fetch blocks that contain only one or just a few internal nodes. We do not deny the fact that after accessing an internal node it may be necessary to retrieve the leaf item, however, this occurs with a much smaller frequency and our serialization engine is not capable of intentionally saving the leaf item in the same block as the parent node item.

### 5.8.1. Cache

As it has been already said the cache may be used at different levels, however the way it is implemented may differ.



The first and most simple type of cache is based on a list or a dictionary that contains a limited number of items. In C# a dictionary is used to represent a collection of keys and values pair of data. When a new item is added and the maximum number of items is reached, one of the items from the cache is removed and sent to the next level.

Another type of cache that we will call layered cache, would remove the item that is least frequently used but this requires a more complicated logic. In some cases it may be required to modify the item that is contained in the cache, as a result it has to be removed from the cache and later inserted once again. This is why it has to implement the search of an item according to the specified key. Such a data structure may be implemented using a queue and a dictionary. The drawback of this solution is the fact that it is not very effective if it is required to remove an element not only by the frequency but also by the key. We cannot locate an item inside a queue without iterating through it. In order to implement a data structure with the required functionality, we decided to use an ordered set of dictionaries. A schematic representation is presented in the following figure.

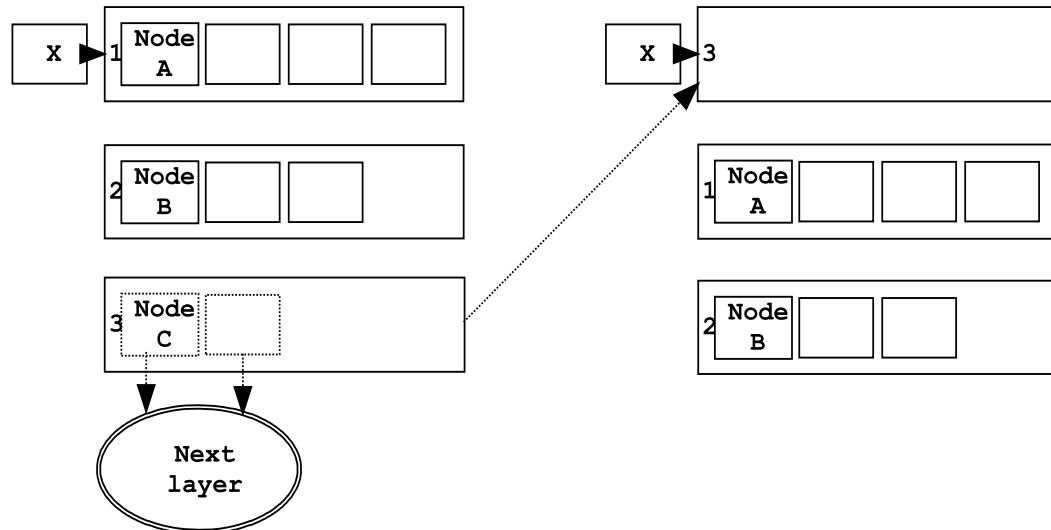


Figure 16 Layered cache insert

The items that are touched (read or written) are inserted in the top dictionary. In our figure this is the item noted by letter X. When the top dictionary reaches the maximum number of items, the bottom dictionary elements are sent to the next level, and the empty dictionary is placed on the top. This is obtained just by reordering the dictionaries. In this way the less frequently used items are sent to the next level when

the cache is full. By saying “full” we mean that the top dictionary limit has been reached.

The levels at which we decided to use one of the two caches are:

- Items – before sending the item to the block, it is saved in a simple cache,
- Blocks – before writing the block to disk, it is saved to the second type of cache,
- Empty block addresses – when the block is deleted, it is more effective not to write the change to the disk, but to use the block while saving another new item.
- Next address – when an item that uses multiple blocks is updated, it is better to use the existing blocks. Normally, in order to find out what addresses have to be used, it would be required to read all the blocks. By using the cache it will be possible to omit the disk block reading phase.

In one of the next chapters we will try to analyse what type of cache to use in order not to waste space.

## 5.9. B+-tree effectiveness

The most effective way would be to build the tree and then to write it to the disk, but it may not fit into RAM. This is why it is mandatory to partially serialise the nodes during the insert operation. It was decided that since the internal nodes are the most frequently used and compared to leaves occupy lesser space, it is better to write them to the disk only when the tree is already built. Also it would be good to limit the maximum number of nodes that will be kept in RAM. Currently the node receives an address in the moment when the first item is inserted into it and it has to keep the address until it is not deleted.

If the internal node occupies an entire block the writing operation would require only one disk access. However, when the node is smaller than a block and it has to be written into a block with other nodes, at first the block has to be fetched and then it has to be written back. Tests have shown that this is very expensive, mainly because the item size is changing and the smart block size is limited. One of the possible solutions would be to give an address to the internal node only in the moment when it is written to the disk, and another solution is to take the biggest

possible required space. The first solution does not completely solve the problem; however, it is better than to create nearly empty blocks, which have to be written and read from the disk. Not only that the second solution will slow down the insertion of further items, but also it will slow down the search operation.

If the item obtains an address only when it is being written to the disk or just disposed, it is necessary to maintain the nodes hierarchy (the full tree branch). This means that it is necessary to maintain in RAM all internal nodes that are situated above the disposed node. It is not possible to dispose the nodes in a random way; otherwise the parent will not contain the address of the child node. However this should not be a problem, because it is impossible to access the child without accessing the parent.

#### 5.9.1. Memory limits

It is expected that the processed datasets will be bigger than the available RAM. This means that the index components will not fit into it. In order to make the indexation more effective we have decided that it is necessary to build each component individually. The order in which this is done does not have any influence on the index structure, because the components are fully independent. The implementation offers two types of indexation, in parallel and in series. In case of smaller RDF datasets the index components may be built in parallel, which is normally faster than the second method. Of course it is important to use the same hash function and program settings.

Next improvement optimises the cache usage. Each cache may occupy megabytes of valuable memory; this is why it is necessary to decide if the usage of each cache is a plus or a minus. As it has already been mentioned, the FullDoc index is one of the biggest; however, the triples do not usually repeat in the same document, it would not make sense. This is why in this case it is unnecessary to have the leaf items in a cache.

#### **5.9.1.1. Node number limit**

The improvements that we have described earlier will definitely be useful mainly because they will lead to an increase of the available amount of RAM; however, they do not solve the out of memory situation. It is important to have the used amount of RAM under control at any moment. This is why it was decided to implement a mechanism that will limit the number of internal nodes. Implementing such a mechanism is another challenge. This mechanism should use the information about the frequency with which the node is accessed (modified). The nodes that are least frequently accessed are perfect candidates to be removed when the maximum number of nodes is reached. This means that a special data structure would be required. We could use the layered cache that has been described earlier, but it is necessary to take into consideration the fact, that when removing a node, we have to remove all child nodes and it may be necessary to update the parent node, because the child node address may change. Also we have to take into account the fact that such data structure has to contain the same number of items as the number of nodes that are in RAM. If the number of items would be smaller, we will not be able to decide which node is the least recently used, because it will be the first candidate for removal when the layered cache will reach its limit. Implementing such a mechanism would be very complicated. First of all, child nodes do not contain any reference to the parent node and the data structure that will maintain all the required information would be too big and very hard to maintain. At each insert it will be necessary to update the information about the nodes that have been accessed.

Another solution would be to remove accessed nodes after inserting a new item. When a new item has been inserted and the number of nodes exceeds the number of allowed nodes, the nodes on the path from the leaf, into which the item has been inserted, until the root will be removed. By removing a node we will also remove all his child nodes. At first sight this may seem to be very ineffective, because frequently used nodes would be removed and then initialised once again. However, this problem should be solved by the block cache that maintains the most frequently used blocks in memory; this means that it also maintains the most frequently used nodes.

It was chosen to implement the second solution that seemed to be much more space effective. Taking into account the fact that the sizes of the internal nodes may differ for every index component, or even just a part of an index component, it would be much more effective to limit the number of nodes using their total size.

In this context it is necessary to describe the way a node is disposed. By disposing a node we destroy the node object and pass it to the next program layer, to a cache or write it to the disk. As a result we decrease the amount of space that nodes occupy. Nevertheless, nodes are also disposed when the indexation process had finished, and the B+-tree is disposed. It has been already said that when disposing a node it is necessary to dispose all its children, as a result, it is important to dispose at first the child nodes. By disposing a child node it may be unnecessary to dispose further nodes, because the total amount of space occupied by the nodes may be smaller than the limit. As a result when disposing a node it is important to take into account the reason why the node is being disposed. In case if we are disposing the B+-tree it is important to dispose all the child nodes and not to stop if the space occupied by the nodes is smaller than the limit.

After performing some tests it was clear that the implemented solution is not good enough for cases when the frequency with which the nodes are accessed is not big enough. For example, branches that have been created in an early stage of index creation process and have not been accessed for a long time, will remain in RAM.

#### **5.9.1.2. *Postponed insertions***

The main problem of the optimisation that has been described in the previous chapter is the fact that when the items are inserted to the B+-Tree, it is necessary to load and unload the nodes that are at a lower level (height). We have tried to process the items in smaller groups that were sorted before insertion, but the problem is that it is impossible to change the way items are inserted, but it is possible to postpone an item insertion. When an item is inserted in a child node that has to be loaded, it is possible to postpone the insertion until the number of items that have to be inserted into one of child nodes is big enough. The idea is fairly simple. Each node will contain a bucket, which will hold the items that have not been inserted yet, because the child node is not loaded. As a result the child nodes will be loaded fewer times,

for the same number of insertions. This mechanism may seem to be easy to implement, but it is mandatory to take into account the fact that the number of postponed items cannot be larger than available space in the parent node. Each postponed item may initiate the splitting of a child item and as a result the number of elements in the parent node will increase. Initially the B+-tree had to be thread-safe [33], offering the possibility of inserting multiple items simultaneously. In order to make it effective and not to lock the full tree, it was necessary to split the child node before moving from the parent node to the child node, otherwise, there was a chance that after inserting a new item the structure of the B+-tree will change and the insertions made from other threads would not be correct. As a result in a postponed insertion it would be illegal to split the parent node after inserting the item in a child node. Also it is important to limit the total number of items in buckets, so that the used memory size is under control, just the same as in the case of the internal nodes.

Once again, tests have shown that the method was not as effective as we have expected it to be. In the moment when it was necessary to empty the bucket, the items from it were usually distributed in different child nodes. The problem was that the child nodes had to be fetched from the disk, and after that, because of excessive memory usage, had to be written back. The buckets would be more useful in case if the child items would not have to be disposed immediately.

#### **5.9.1.3. LRU items**

The next and the last improvement that was proposed and implemented is the data structure that would maintain the information about the least recently used nodes. The idea is very similar to the one presented earlier but which has been rejected because it was too complicated. Initially we were thinking that it is necessary to maintain the information about the usage frequency for each node. However, in order to be able to decide what tree branch is least frequently used; it is enough to define the distance from the root (the level) and to maintain the LRU information just for this tree level. As a result we may calculate the maximum number of nodes for which we will be keeping the LRU information.

The data structure that will offer the LRU information will be implemented as a modified version of the principle that has been described in the layered cache. The

LRU data structure will be formed by a set of dictionaries, which will have a limited size. The schematic representation is shown in the following figure.

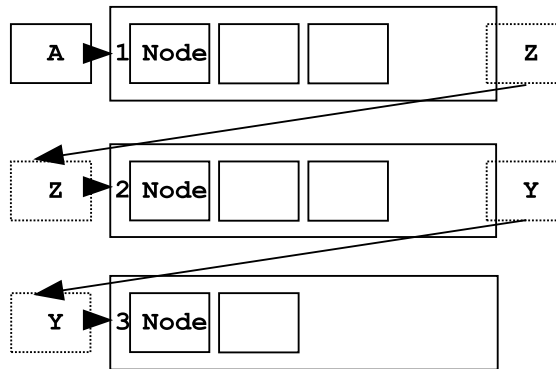


Figure 17 LRU structure inserting a new item

The dictionaries will be ordered, and the lowest dictionary will contain the least recently used node. Each time a node is accessed it is located inside the LRU data structure by searching each dictionary, after which it is inserted in the upper dictionary. During this process it may be necessary to remove a node from the dictionary into which the node has been inserted. The removed node has to be inserted in the same manner into the next dictionary, which is at a lower level. It is important to notice that the dictionaries are not allowed to change their capacity or position. Trying to insert a higher number of nodes than the capacity, should be illegal. From the description in which the LRU data structure is used, it is clear that we can define the required capacity; as a result, by creating a LRU data structure with a bigger capacity than the number of items at the specified tree level is a mistake.

We decided that the best tree level for which we will be keeping this information is the second level, the root being at the zero level. As a result the maximum number of nodes in the LRU data structure would be  $n^3$ , where  $n$  is the number of items in a node.

While using the LRU data structure it is important to take into account that the nodes may change the level at which they are located inside the B+-tree. This may happen after splitting the root node. As a result the LRU data structure will not contain relevant information. The LRU data structure has to contain all the nodes that are located at the second level and if the level of one of this nodes will change, this

means that the LRU data structure will have to hold  $(n^3) + 1$  items. As a result, a valid second level node will be removed from the LRU data structure without being disposed, in favour of a node that is not located at the second level. Also it may lead to the situation when the node that is situated on the third level will be returned as the least recently used node. As a result, the tree branch may be disposed even if it is one of the most frequently used. Another scenario is the increase of the LRU data structure capacity, which is also undesirable, because this may lead to the fact that we will lose control over the used amount of memory.

Each time a node changes its location, from level two to level three, it is necessary to update the LRU data structure, by removing the nodes that are in the third level. When the root is split, all the nodes from the second level are moved to the third level, which means that it is necessary to remove all the items from the LRU data structure. Nevertheless, if we remove all the items, we will not be able to get the required least recently used node from LRU data structure. Each time a node is inserted into the LRU data structure it is necessary to verify if it is contained in one of the dictionaries. In case if the node is missing it is necessary to remove all the child nodes from the LRU data structure.

## 5.10.RDF parser and SPARQL engine

The implementation is divided into three main layers: the RDF file parser, the SPARQL query engine and the RDF index itself. Developing a RDF file parser and a SPARQL query engine is beyond this project. We decided to use one of the existing libraries; still, the number of such libraries that have been developed for .Net is not so big, and we had to use the SemWeb project. After making a few fixes to the RDF parser we were able to process the required files. The only part that we could not change was the SPARQL engine which uses a Java library. We will describe the reasons why we wanted to modify it in a later chapter.

We have already explained the structure of the RDF index and it remains to describe the last two layers. As it has been already said, we have chosen to use an existing library; as a result parsing the RDF file is very easy. We just have to provide a method that will process a triple and will prepare it for the indexation process. The



usage of the SPARQL query engine is a little more complicated, but at a higher abstraction level it is enough to set the method that will process a simple query that is described by an internal object. The internal object contains: the known fields values, the unknown fields and the number of required solutions. Each known field is represented as a list of known values. In order to answer to a simple query it is necessary to find all triples that match at least one value from each known field. At this stage it is required to provide the textual representation of the unknown fields; as a result the triples have to be fetched from the FullDoc index. In the testing chapter we will explain on a query why this technique is not very efficient.

In the end we would like to mention, that we could not find any other library that would fit better our requirements, and writing our own SPARQL engine is a complicated task that cannot be solved in this project.

## 6. RDF index testing

In this chapter we will try to present the results of the indexation process and what is more important, we will show the SPARQL benchmark results. In order to obtain relevant results we have decided to use one of the existing benchmarks. The most frequently used benchmarks are:

1. Berlin SPARQL Benchmark (BSBM) [34]
2. Lehigh University Benchmark (LUBM) [35]
3. Social Network Intelligence Benchmark (SIB) [36]
4. DBpedia SPARQL Benchmark [37]

After analysing the documentation, we have decided to use the BSBM V3. This benchmark has already three versions and it offers a Java tool that generates RDF files of any size. It uses three dictionaries that contain product names, names of persons and random words from English text. The dataset may be generated using one of the three representations: RDF triple, named graphs or relational. Nevertheless, the parser that we have decided to use supports the RDF triple representation. The Java tool has been added to the attached DVD and the documentation may be viewed on the official website.

The used SPARQL queries are built around real life use cases. They simulate a consumer, which is looking for a product according to some criteria. The benchmark contains 12 queries and each query has a set of properties that are as well described in the documentation. One of the minuses of using such a benchmark is the fact that the same subject, predicate or object may occur a few hundreds or even thousands of times. This may be considered unusual for real life cases and may complicate the indexation process.

The provided queries are designed to access a large amount of RDF data. In the documentation it is noted that an increasing number of Semantic Web applications do not rely on complex reasoning but focus on the integration and visualization of large amounts of data.

The next important step is to choose the projects with which we will compare our index. After analysing the information from previous chapters we have decided to compare our index with the Sesame and Jena indexes, which are based on native triple stores. We did also want to compare our index with the dotNetRDF<sup>7</sup> library, which would create a RDBMS index. We have tried to use it before our index has been implemented. However, we did not manage to create the dotNetRDF index for a dataset that had more than 100 000 triples and we have decided not to proceed further.

## 6.1. Dataset specification

In order to be able to understand the structure and the meaning of each query, we have to start by describing the structure of the used data. The dataset contains the following classes:

1. Product
2. ProductType - products are organized in hierarchies and each product is in one of the hierarchy leafs. The depth of the hierarchy depends on the number of products and varies from 2 to 6.
3. ProductFeature – the set of product features depends on the product type and its super classes, as a result some features are very generic while others are more specific.
4. Producer - each product has a producer, which is generated for 50 products on average.
5. Vendor
6. Offer – products may have multiple vendors and each offer belongs to a vendor.
7. Person – (review author) each review has a person, which is considered to be the author.
8. Review – each product has on average 10 reviews.

The used namespace are enumerated in the following table:

---

<sup>7</sup> <http://www.dotnetrdf.org/>

Prefix	Namespaces
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
foaf:	http://xmlns.com/foaf/0.1/
dc:	http://purl.org/dc/elements/1.1/
xsd:	http://www.w3.org/2001/XMLSchema#
rev:	http://purl.org/stuff/rev#
bsbm:	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
bsbm-inst:	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/

Table 2 Benchmark namespaces

By analysing the following figure it is possible to understand the structure of each class and the relationship between them.

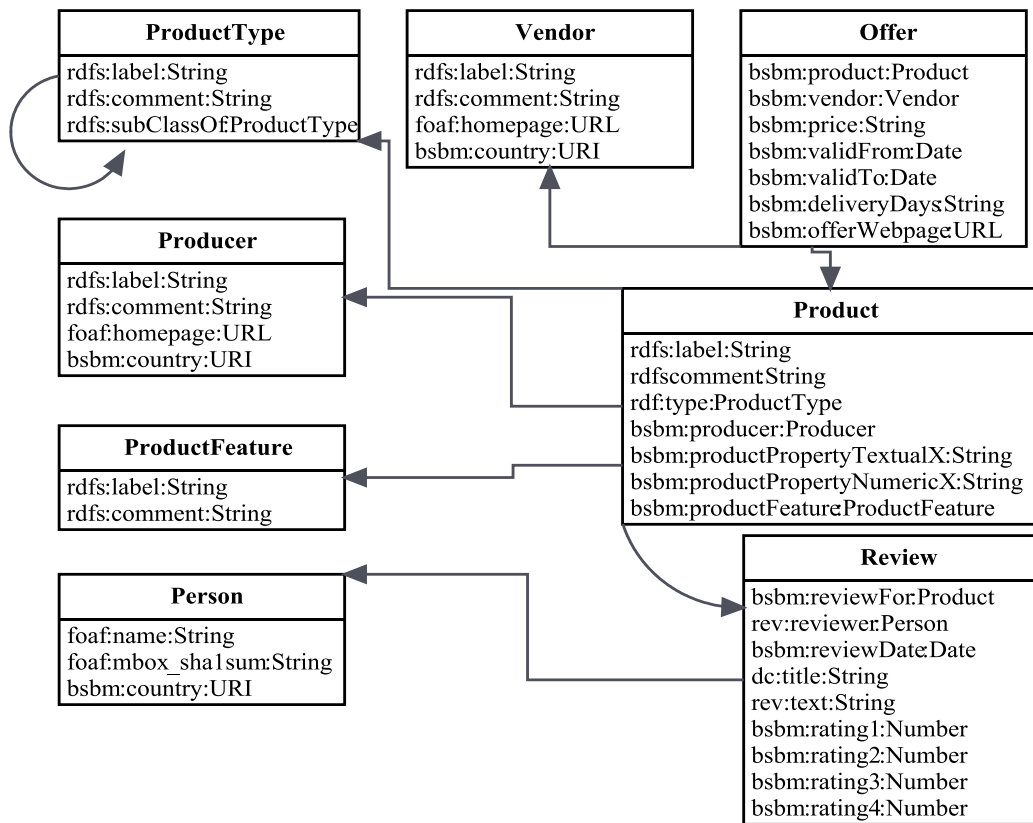


Figure 18 Berlin benchmark dataset structure

We may see that the dataset contains all kinds of data types: numbers, texts and dates. The most interesting for us are the text attributes, which according to the documentation may be 50 to 200 words long. This is very important because in such way real life cases are simulated.

## 6.2. Indexation

The first part of the tests consists in creating the index. In order to offer a better analyse we have decided to run the indexation using three RDF datasets of different sizes: 1, 10 and 25 millions of triples. As we will see further, this is necessary in order to prove the correctness of the tests. We cannot guarantee that the other indexes or even our index is capable to index RDF datasets of the 25 or even 10 millions of triples. This is why we start by successfully indexing small datasets and proceed to bigger one, until the indexation process fails because the dataset is too big. At first we will analyse the indexation process of our index and in the end we will display in a table the results of the other two projects. We will illustrate some of the prepared graphs and we will explain what is happening during the indexation. We will also try to show the effectiveness of the used caches.

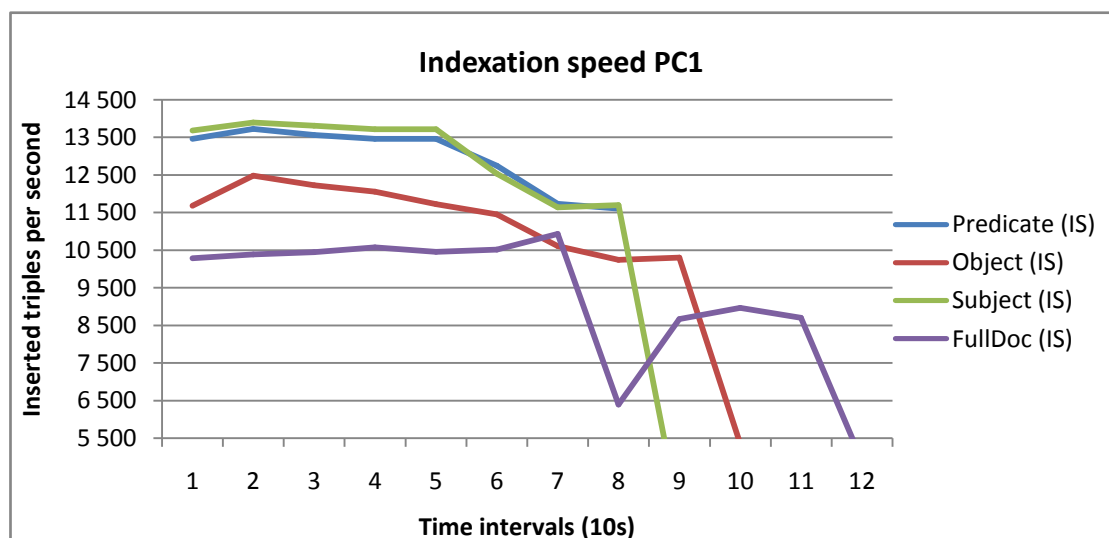
Throughout the tests we have used two different computers, which in our opinion may be considered to be in the category of home or office PCs. The first has an Intel Core Quad CPU 2.83 GHz and 3.25 GB of RAM; a disk that runs up to 7200 rpm and it uses a 32 bit Windows XP OS. We will call it PC1. The second computer, which we will call PC2, has an Intel Core 2 Duo CPU 2.93 GHz and 3.46 GB of RAM; a disk that runs up to 7200 rpm and it uses a 32 bit Windows 7 OS. Latter we will explain why we have decided to illustrate the test results for these two computers.

Beside the index itself, during the indexation process it is possible to create the statistics files, which are in the CSV format and are created in the same directory as the index. The information from these files may help to optimise the index structure and find the bottlenecks. Each index component offers its own statistics file that contains the information about: the number of written or read blocks according to their type, the number of inserted items that have not been present in the index or the number of items that have been inserted but were already in the index, the statistics that are related to the buckets, the size of the indexed triples etc. In the provided excel sheets we have included the above mentioned statistics. The columns are described in the appendix D.

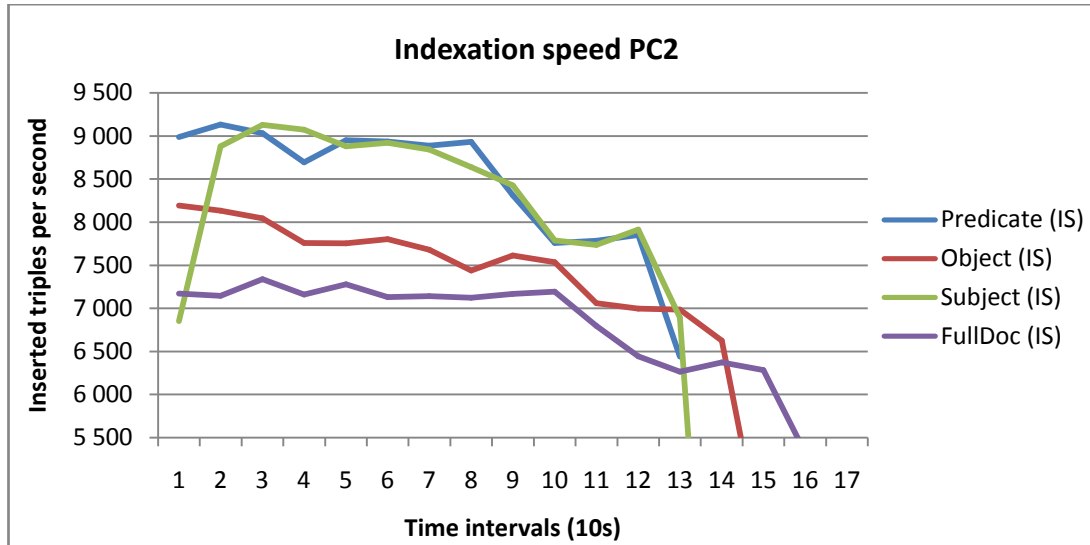
Before proceeding, we would like to clear some frequently used expressions. The block request is the process of initialisation of a block of data from any available resource; this may be a cache or the disk. The disk request is the process of initialisation of a block of data directly from the disk or the operating system cache. By disposing a block we destroy it and the data are sent to a system cache or are persisted to the disk, in order to restore it, we have to send a block request.

### 6.2.1. One million triples

The following graph displays the indexation speed for each index component for the 1 million triples dataset. We have measured how many triples have been inserted into each index component. The measurements have been repeated in intervals of 10 seconds until the indexation process has finished. The vertical axe represents the number of triples that have been indexed in an interval of 1 second. The horizontal axe represents the total number of time intervals. We may notice that each component has a different indexation speed, and this is one of the reasons why it is more efficient to build the index sequentially. If we would build the index in parallel, the slowest index component will also slow all other index components. Furthermore, from the graph we see how with the increasing number of inserted triples, decreases the indexation speed. Taking into account the fact that our index is based on a modification of the B+-tree, we were expecting at least a logarithmic speed decrease; nevertheless, using the provided statistics we will try to explain the real factors that influence the indexation speed.



Graph 1 Indexation speed PC1 (1 million)



Graph 2 Indexation speed PC2 (1 million)

We may see that the first PC offers a higher indexation speed. At first we thought that it may be caused by a more powerful processor, or a faster hard disk. We will try to find an explanation, when we will analyse the indexation process for bigger datasets. The second PC has a less stable indexation process. Taking into account that we have indexed the same dataset, we believe that the speed fluctuation is mainly caused by the fact that the time required to build the index components is longer and the number of statistical measurements is higher.

Besides that, we may notice other smaller speed fluctuations in both graphs. Initially we considered that it could be explained by the way in which the cache is working. When it is full, we have to write a big amount of data to the disk (flush the cache) as a result triples are not indexed. Nonetheless, the statistics have proved that the cache is not flushed, since the amount of data is small enough to fit into it. Another possible explanation may be the fact that smaller triples are indexed with a higher speed. This does not have to be a rule, but we have noticed that the triples, which are located at the end of the dataset, are formed by longer literals. From the statistics, which offer us the information about the number of processed characters, we may deduce that the size of the indexed triples increases by approximately 30% and does not decrease until the indexation is finished. We have measured the speed with which the RDF parser provides with data the indexation engine and have observed that it has the same speed decrease. As a result, another important factor that may influence the indexation speed is the dataset triples size. Also in the end of

the indexation process, all the internal resources are disposed and unsaved or modified nodes are being persisted to the disk. The number of indexed items may be very small, while the number of write operations will be very high. Using these observations we can tell for sure that the biggest speed decrease, in the last part of the graphs, is determined by the triples size and by the high number of writings to the disk.

Another test has shown that the time required to read the triples from the dataset and to transform them to hashed values may constitute up to 50% of the indexation period<sup>8</sup>. Nonetheless, the reading of the triples by the RDF parser takes up to 10% of the indexation period.

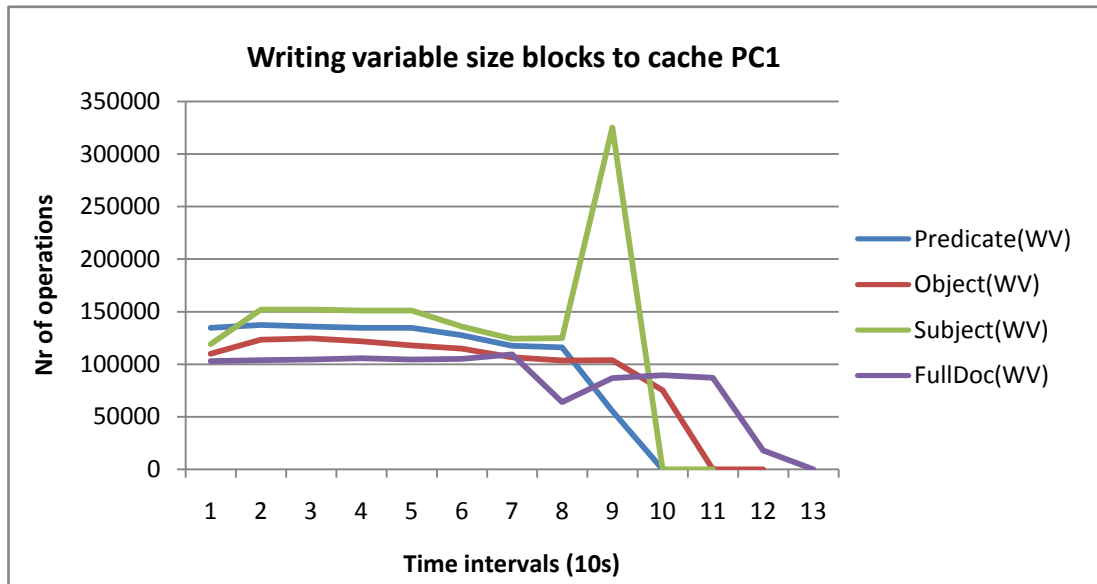
We may also notice that the FullDoc index has a strange speed decrease but latter the speed increases. Based on the provided statistics, we believe that this is the result of the fact that a new data file has been created. The FullDoc index component has the lowest speed, which can be explained by the fact that it is normally writing a higher amount of data.

Other interesting information is the number of accessed disk blocks. In order to find the bottleneck of the indexation process we have decided to measure how many readings and writings are performed. The next two graphs display the intensity with which data are written to variable size blocks. The variable size blocks contain the items from the B+-tree leaves, which are one of the biggest. Nevertheless, the frequency with which they are accessed may differ for each index component. A high number of operations may mean two things: that the indexation process is fast and effective, or that the number of accessed variable size blocks is much to high. If we compare these graphs with those that have been presented earlier (indexation speed), we may notice that they have quite similar dropdowns. This means that the speed decrease is not directly influenced by the increased number of operations.

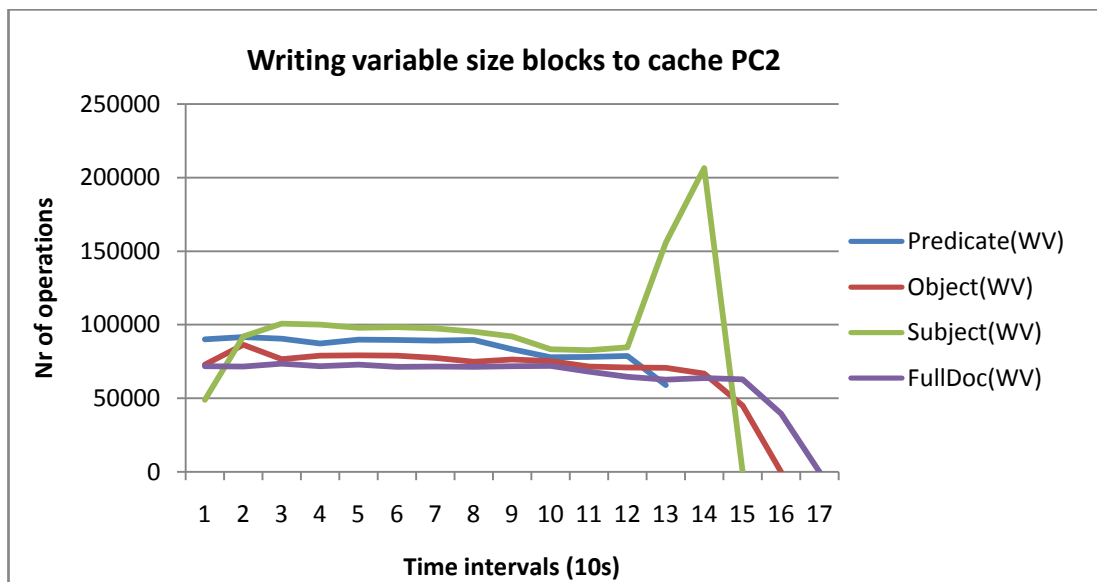
---

<sup>8</sup> The test has been performed only on one computer using a dataset of nearly 11 millions of triples for the subject index component.





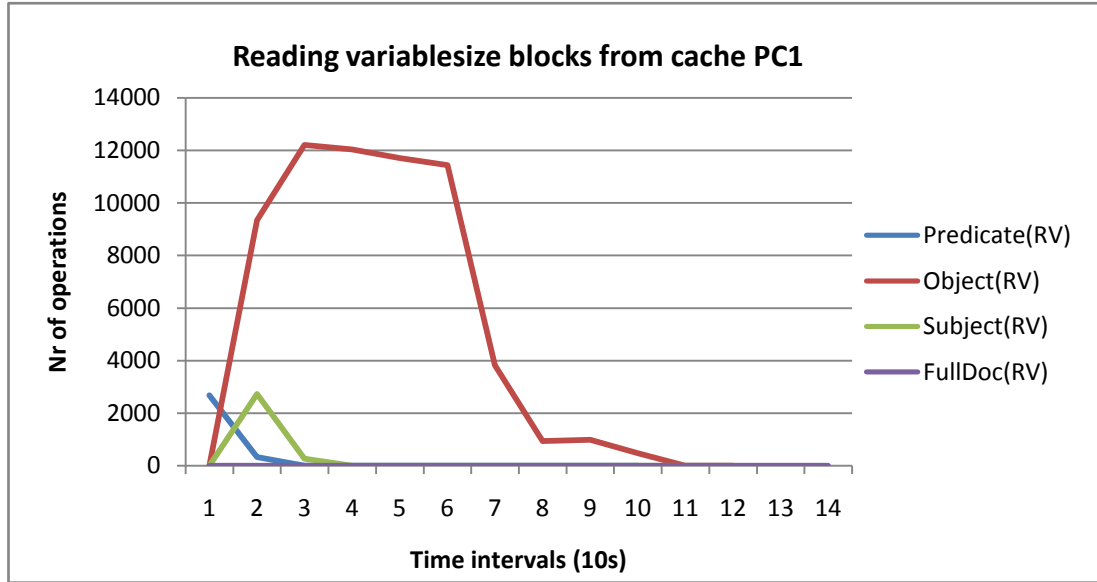
Graph 3 Writing variable size blocks to cache PC1 (1 million)



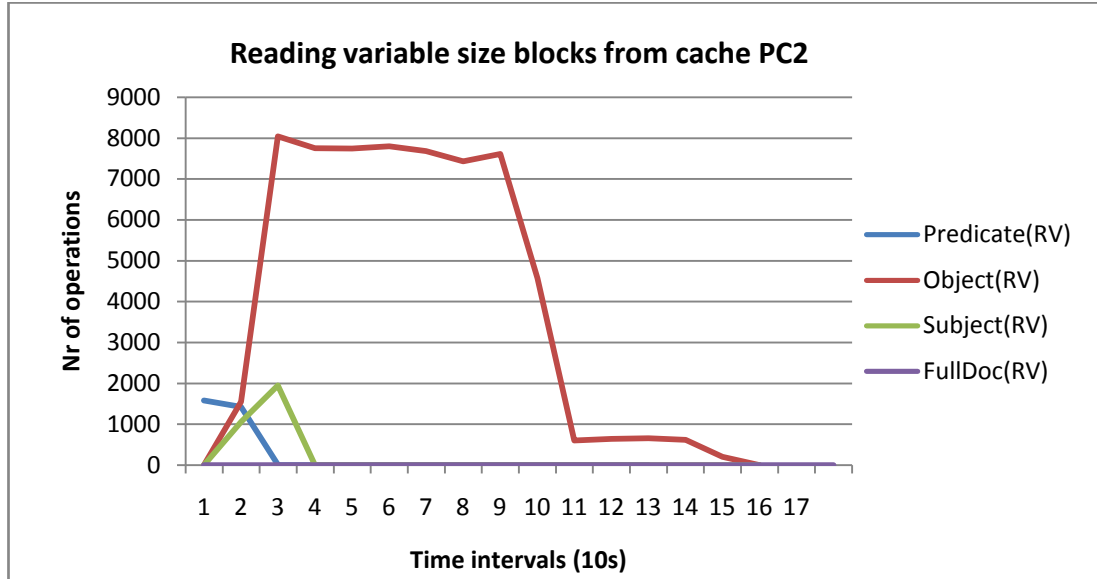
Graph 4 Writing variable size blocks to cache PC2 (1 million)

The next two graphs display the number of read operations for the variable size blocks. We may notice that the two graphs are very similar, but there is a big difference between the object index component and all other index components. After analysing the triples and the statistics, which describe how many of the inserted items are or not present in the index at the insertion moment, we have come to the conclusion that during the creation of the object index component, the sub-tree contains a higher number of keys (subjects) that are already present in the index. As a result, it is necessary to reload multiple variable size blocks. In order to minimise the number of read operations for the object index component, we have tried to

modify the structure of the sub-tree, by swapping the subject key with the predicate key. Nonetheless, the number of predicates was a lot smaller than the number of subjects and the results were even worse; therefore we had to undo the changes. The fact that the number of read operations is so high may be a problem when indexing bigger datasets.



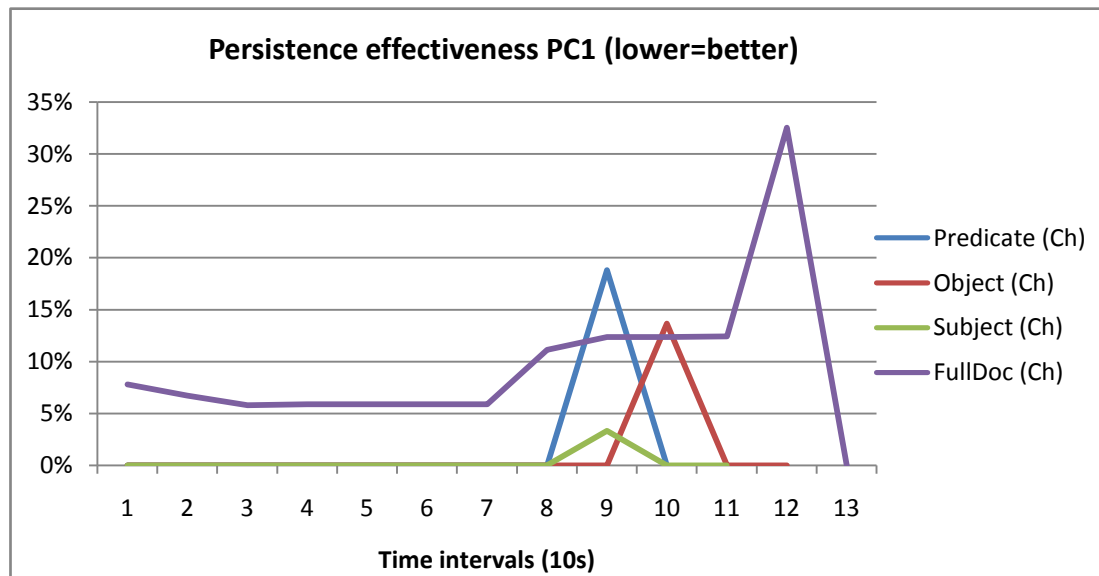
Graph 5 Reading variable size blocks from cache PC1 (1 million)



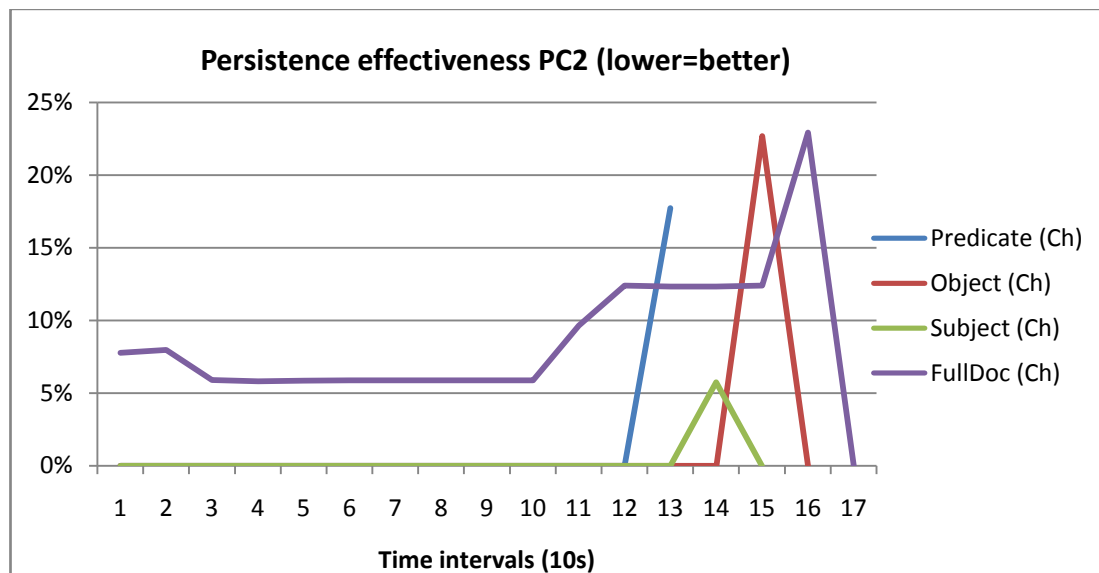
Graph 6 Reading variable size blocks from cache PC2 (1 million)

Next we will describe the effectiveness of the implemented persistence model and the block cache. In order to analyse it, we have measured the total number of read and write operations, and the total number of disk requests. The number of operations is much higher than the number of disk requests, because one block may

contain multiple items. This is why we will determine the dependency between the performed operations and the disk requests, by dividing the number of disk requests to the number of operations. The value will represent how many percents of the operations have raised a disk request. A lower percentage means fewer disk requests and is the result we want to obtain. Of course it is also important to take into account the total number of operations. A small number of operations combined with a low number of disk requests may as well result in a low percentage.



Graph 7 Persistence effectiveness PC1 (1 million)



Graph 8 Persistence effectiveness PC2 (1 million)

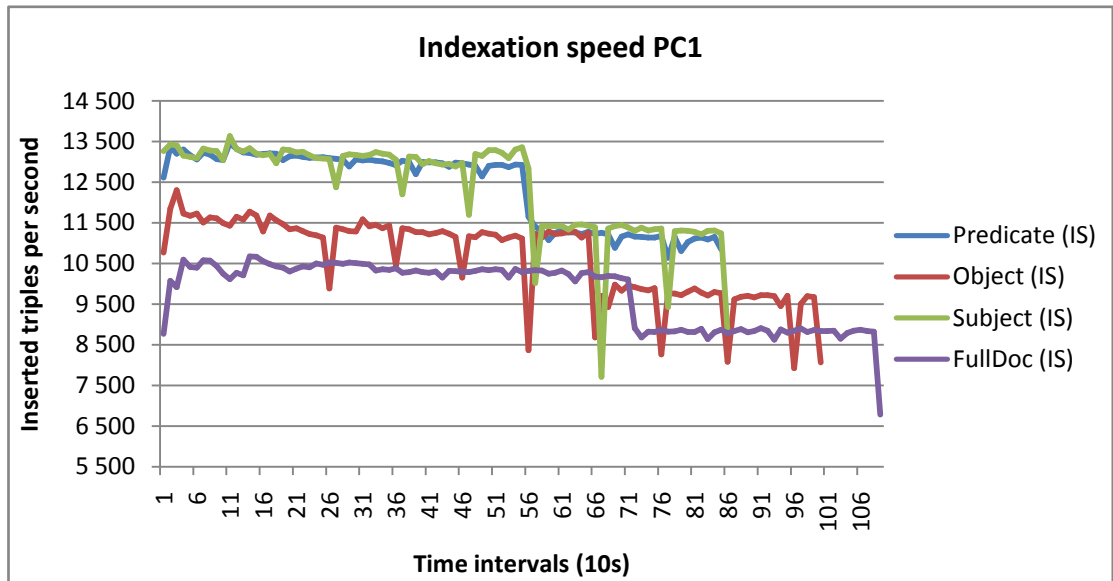
From the above two graphs we may see that the highest number of operations has occurred in the last 25-35 % of the indexation process. The lack of

operations in the yearly stage of the indexation is explained by the fact that the cache is not full and effectively fulfils its task. We may notice the opposite in the case of the FullDoc index component, which does not use the cache. The early increase of the disk operations in the case of the FullDoc index component may be explained by the fact, that in the end of the dataset the triples size is much bigger. As a result it requires more space while being persisted to the disk. The increase in the case of the subject and predicate indexes is explained by the fact that the file block cache is flushed and data are persisted to the disk.

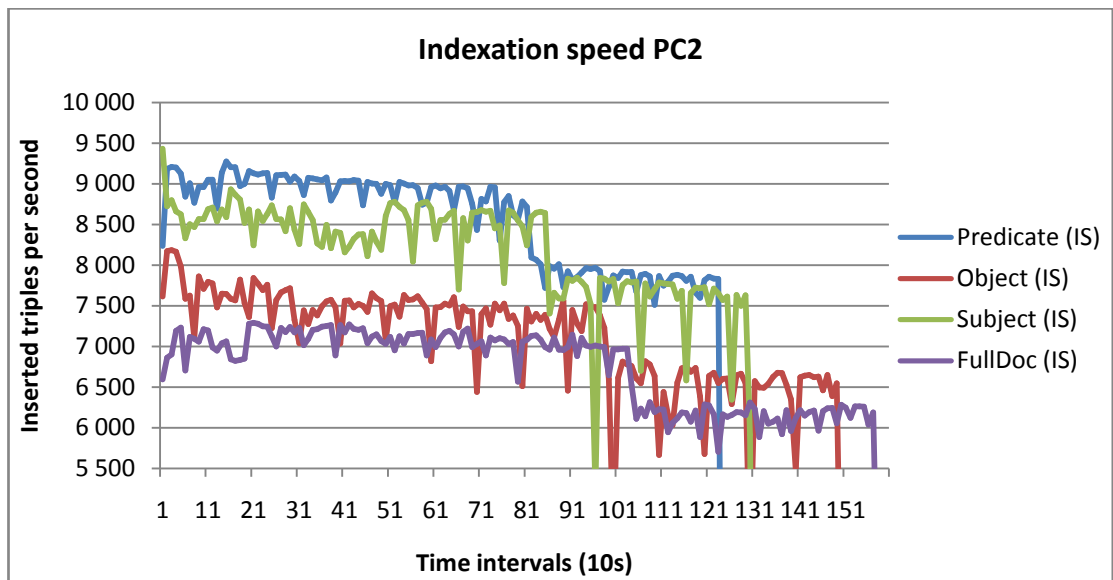
In the attached Excel files, “3000\_PC1.xlsx” and “3000\_PC2.xlsx”, we have presented the full indexation results. Besides the information presented in this paper they also contains information that has been used during the implementation phase.

#### 6.2.2. Ten millions triples

Next we have built the index for the 10 millions triples dataset. In the following graphs we may see the indexation speed, which is comparable with the indexation of a 1 million triples dataset. The full indexation took approximately ten times longer, but in the same time the dataset is ten times bigger. On both graphs we may notice regular dropdowns of the indexation speed. We have performed a variety of tests in order to find out the cause. At first we thought that it may be caused by the fact that one of the caches is flushed or it may be necessary to allocate memory for it, but the statistics proved this to be wrong. Next we checked if the RDF parser has any regular speed dropdowns. Nevertheless, running the indexation without any insertion into the index proved that the RDF parser does not have any regular speed dropdowns. Another hypothesis was that the OS is flushing its internal caches in order to write the data to the disk. We performed a test during which the writing to the disk was omitted, but the graph was still showing similar speed dropdowns. Unfortunately we could not find any provable explanation.

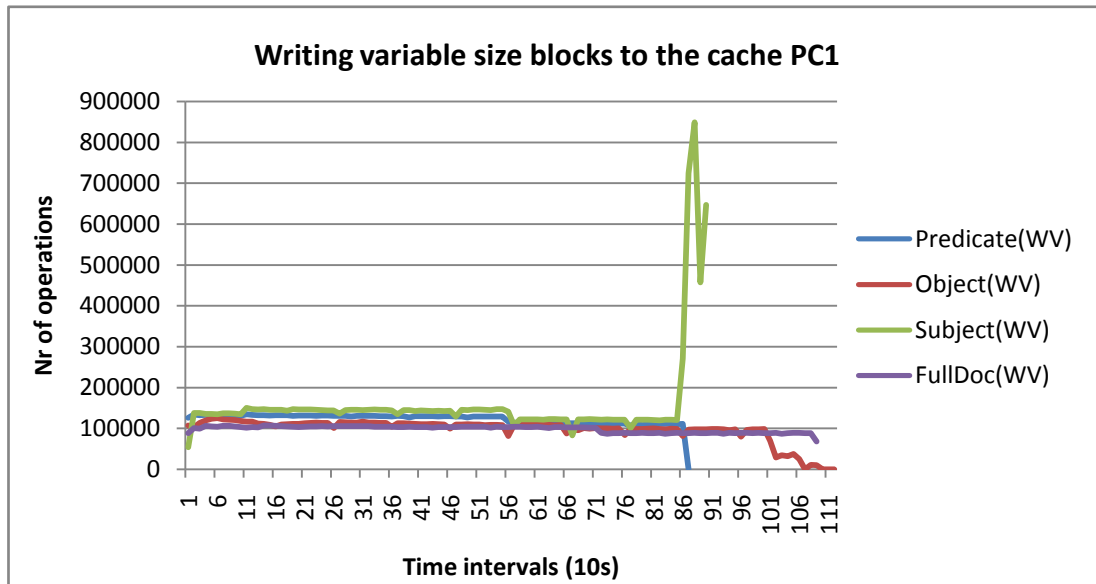


Graph 9 Indexation speed PC1 (10 millions)

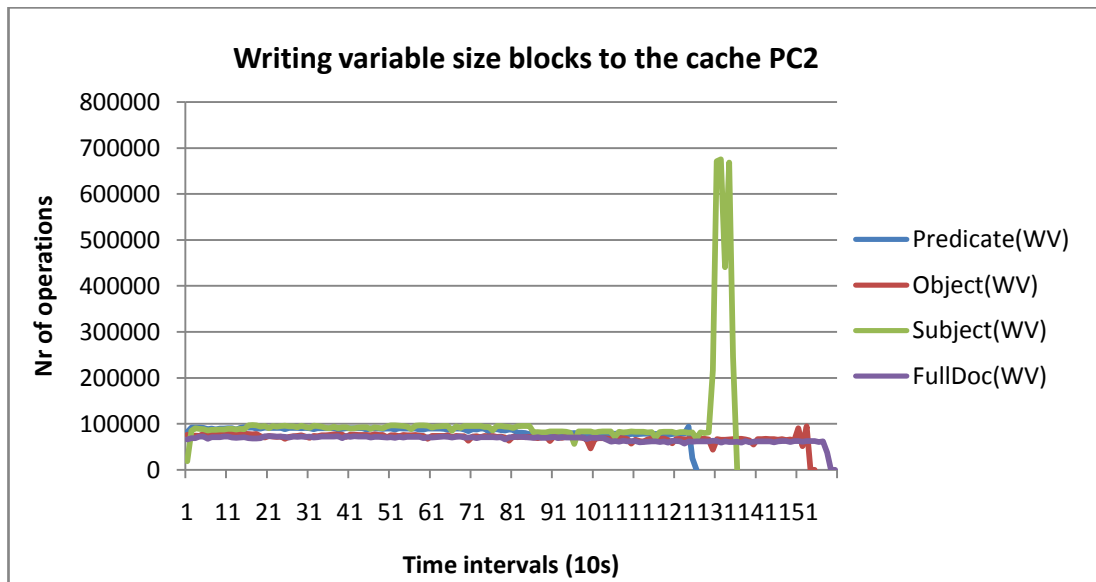


Graph 10 Indexation speed PC2 (10 millions)

Next we will present the intensity with which variable size blocks are written to the cache and to the disk. From the following graphs we may notice that the number of operations does not show any major fluctuations, except the increase in the end of the indexation process. In the case of the FullDoc index component this is explained by the increased size of the triple literals. In the case of other index components this is explained by the need to flush all catches before finishing the indexation process.

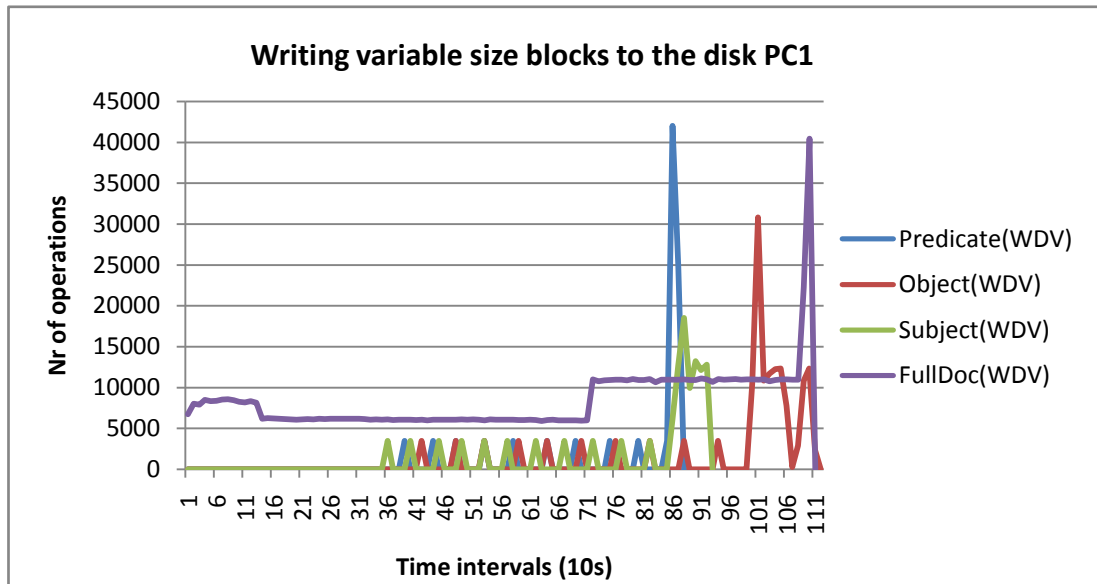


Graph 11 Writing variable size blocks to cache PC1 (10 millions)

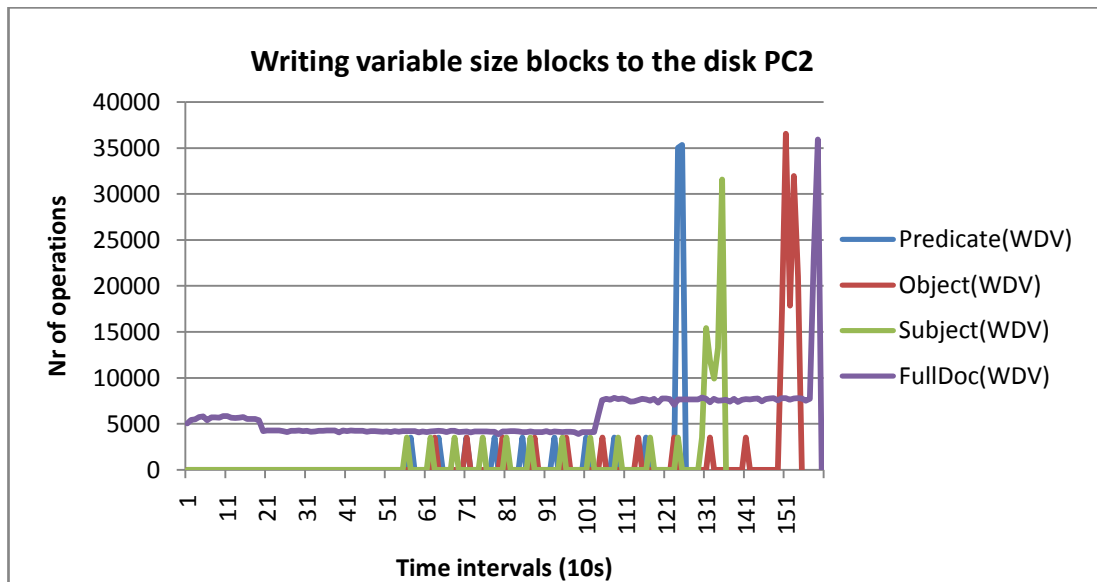


Graph 12 Writing variable size blocks to cache PC2 (10 millions)

We may also notice that the number of operations that involve the cache is a lot bigger than the number of operations that involve the disk. This means that the cache fulfils its task and we managed to decrease in this way the disk usage frequency.

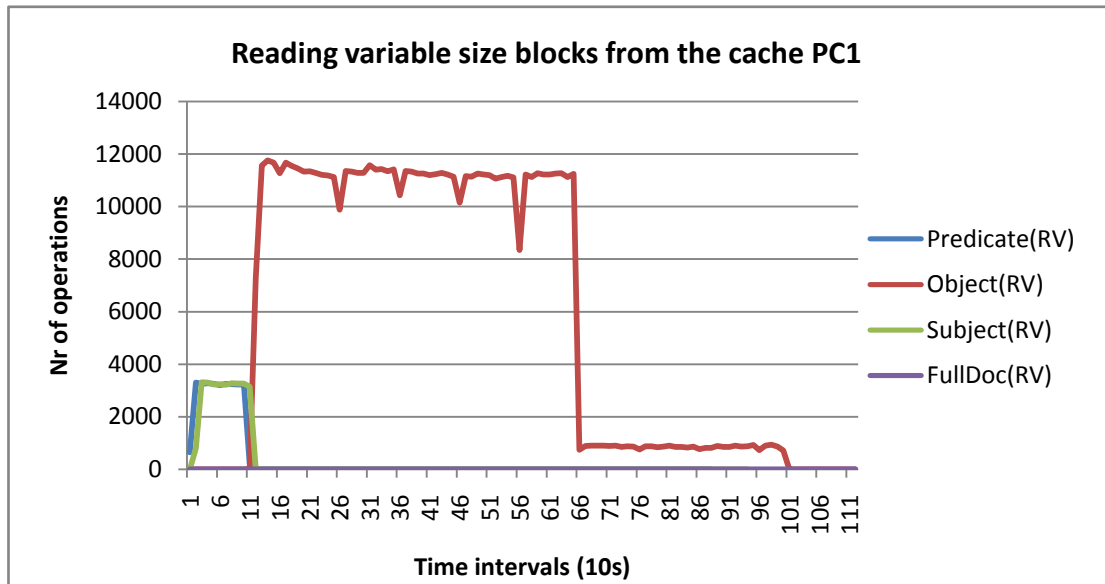


Graph 13 Writing variable size blocks to disk PC1 (10 millions)

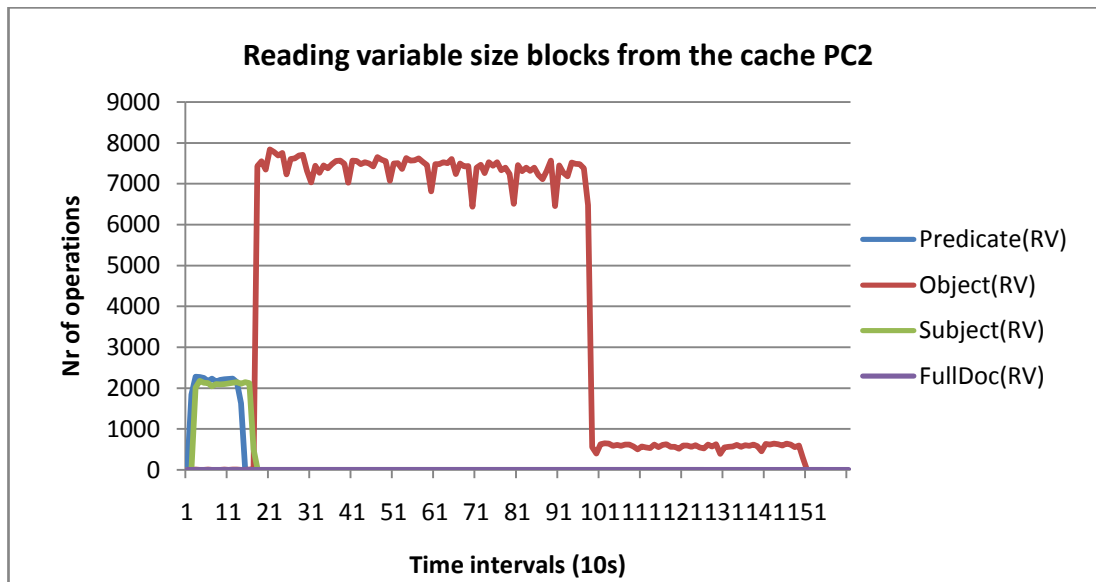


Graph 14 Writing variable size blocks to disk PC2 (10 millions)

The graphs that corresponds to the number of readings of the variable size blocks from the cache have a mildly similar shape as the one that have been presented for the indexation of 1 million triples dataset. Once again the object index component is overusing the variable size blocks cache.



Graph 15 Reading variable size blocks from cache PC1 (10 millions)

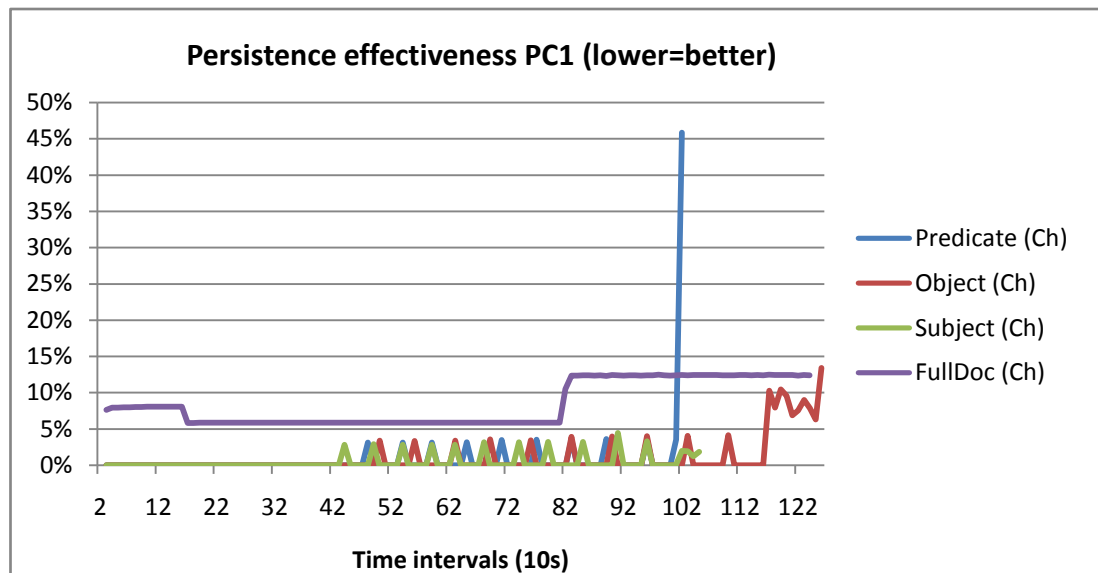


Graph 16 Reading variable size blocks from cache PC2 (10 millions)

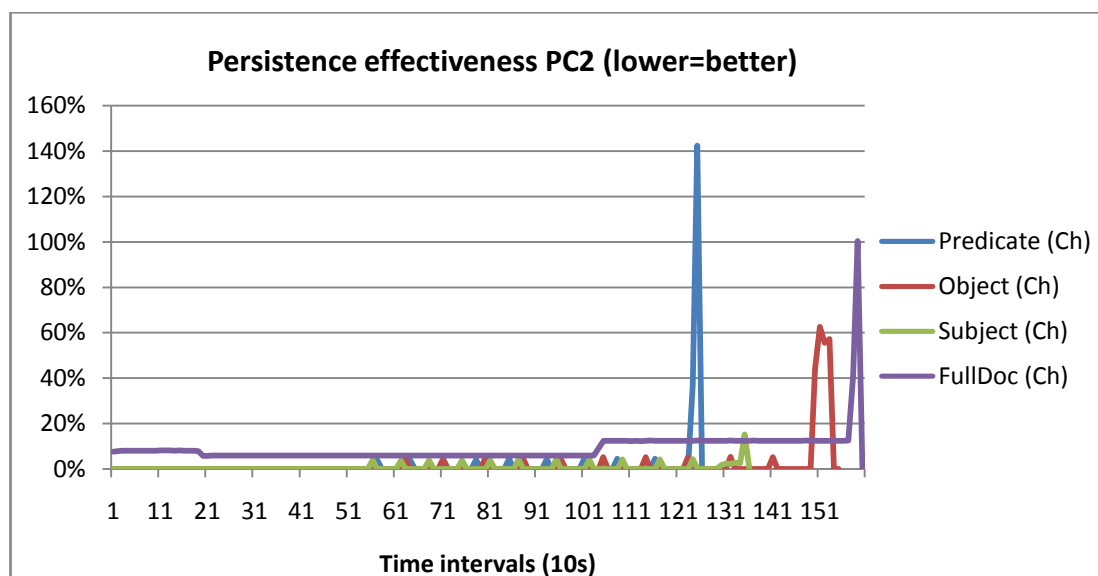
Next we will present the persistence effectiveness graphs. We may notice that the graphs have similar shapes as those that have been presented earlier, when we were describing the writing of variable size blocks to the disk. The only difference is the shape of the subject component index line. We may notice that the flushing of the cache does not change the persistence effectiveness. This is explained by the fact that the number of operations that use the cache is higher than those that use the disk. Based on the statistics we may say that the number of modified variable size blocks is 3-6 times higher than normally. In the same time, the number of variable size blocks that are fetched from cache is equal to 0. This means that the cache is not



overused. The high number of modifications of the variable size blocks is caused in the moment when the subject index component is persisted. The size of the nodes from the sub-trees is small; as a result it is necessary to fill one variable size block with multiple nodes.



Graph 17 Persistence effectiveness PC1 (10 millions)



Graph 18 Persistence effectiveness PC2 (10 millions)

After analysing all the graphs we may say that the behaviour of the indexation process for the 1 and 10 millions datasets may be considered very similar. In the attached excel files: “30000\_PC1.xlsx” and “30000\_PC2.xlsx” we have presented the full indexation statistics for the 10 millions dataset.

### 6.2.3. Twenty five millions triples

The creation of the index for the 25 millions triples dataset was a real challenge for us. As we have predicted earlier, the biggest problem was the creation of the object index component. For both PC's, the time required to build the object index is up to 2-2.3 times longer than compared to time required to build other index components. This is explained by the high number of operations during which the variable size blocks are accessed. Unfortunately, as we have mentioned earlier, our implementation does not provide the necessary flexibility in order to be able to fix this problem. We believe that a solution could be the implementation of a sub-tree that will be referenced in the leaves of the existing sub-tree. As a result, the list that will be contained in the leaves will hold only the FullDoc key.

The amount of generated data is too big this is why the indexation statistics may be found in the provided Excel tables: “70000\_PC1.xlsx” and “70000\_PC2.xlsx”. From the presented statistics we may deduce that the indexation speed decreases in the moment when it is necessary to reduce the number of B+-tree nodes that are maintained in RAM. Nonetheless, the number of removed nodes is higher than the number of loaded nodes. For example in the case of the FullDoc index component the number of removed nodes varies from 15000 up to 50000 per second, while the number of read blocks varies from 1000 up to 2500 per second. In the case of other index components the difference between these two values is smaller; however the number of removed nodes is usually smaller than the number of loaded nodes.

### 6.2.4. Indexation results

In this chapter we will present the indexation results compared to the other two projects. The next table illustrates the time required to build the index on the first PC.

Number of triples PC1	Our index	Jena	Sesame
1 million	393 sec	151 sec	599 sec
10 millions	67 min	---	412 min
25 millions	4,8 hours	---	10++ hours

**Table 3 Indexation speed PC1**

As we have mentioned earlier, it took us nearly 5 hours to build our index for the 25 millions dataset on the first PC. Nonetheless, neither did the Sesame project managed to complete the task. We had to stop the indexation process after 10 hours. It is not clear why the Sesame project was so slow. The Jena project was even worse. We were not able to finish the indexation of the 10 millions triples dataset. Each time we tried to index it, the program stopped working because of a heap memory exception. After searching for the solution, we have found out that this is a known issue that occurs on 32 bit platforms. On such a platform Java virtual machine may use for the heap at most 1500-1600 MB. In order to be able to use the Jena index in the next evaluation phase, we have decided to use a different computer, which would run a 64 bit Java version. Even so, the Jena index could not be created because of a GC overhead limit exceeded exception. Unfortunately we could not find any solution and decided to continue the evaluation only with the Sesame project. We have to say that during the implementation phase, we have noticed that our index may also use at most 1600 MB of the RAM.

In the next table we may see the results, which have been obtained on the second PC. From the results of the Sesame project we may say that the second PC configuration is more appropriate for the performed task. However, we cannot say the same about our index, which if compared to the results obtained on PC1 runs slower on PC2.

Number of triples PC2	Our index	Jena	Sesame
1 million	564 sec	80 sec	181 sec
10 millions	96 min	---	49 min
25 millions	7 hours	---	15++ hours

**Table 4 Indexation speed PC2**

The next table displays the index size on the disk. We may notice that our index is the biggest, and this may be one of the reasons why it requires more time in order to be built. We have tried to analyse the structure of our index data files, but we did not notice any big amounts of empty space. The size may also be an issue during the SPARQL queries testing phase, mainly because our index will probably have to fetch from disk a bigger amount of data.

Number of triples	Our index	Jena	Sesame
1 million	504 MB	415 MB	220 MB
10 millions	4.86 GB	---	2.13 GB
25 millions	11.5 GB	---	---

Table 5 Index size

In the end we would like to mention that we have built the indexes in series as it has been described in the memory optimisations chapter. Nevertheless, we have tried to build the index in parallel for the 1 millions triple dataset. On the first PC the task has been finished in lesser than 250 seconds while on the second PC it was built in 400 seconds. This type of indexation is faster but may be used only when the dataset is not too large. We did not perform other experiments that would explain why there is such a big difference, but it would be interesting to build the index in series, without the usage of the hash function. The hash may be computed during the creation of the FullDoc index and written to a file, while all other index components will just read the values from that file.

### 6.3. SPARQL Queries

In the following sections we will describe each query and will present the benchmark results. The queries have been evaluated on an Intel Core i3 CPU 2.27 GHz, 3.86 GB of RAM, a disk that runs up to 5400 rpm and it operates on a 64 bit Windows 7. Further we will present the SPARQL queries and in order to make them shorter we will omit the used prefixes (namespace), which have been enumerated earlier. The prefixes have to be declared before the query itself in the following way:

```
PREFIX bsbm: <http://www4.wiwiiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

The query evaluation time will be presented for each query in a separate table in the `minutes:seconds.milliseconds` format. Initially we wanted to present the results for the following execution types:

- Fresh start – the index has never been used before, as a result it is not possible to use the cache.
- Multiple runs – the query is executed multiple times, in order to find the best execution time.
- Query mix – the query is evaluated after evaluating other random queries. In this case it is possible that a part of the solution will be already contained in the cache.

After running a few tests we have come to the conclusion, that it is too hard to give relevant results. The first execution type is nearly impossible to simulate without restarting the PC, otherwise there is a big chance that some of the system cache may alter the results. The results for the last two execution types are very often equal. Usually the difference is just a few milliseconds, which may be caused by the influence of multiple factors that are not related to the index speed. As a result we have decided to run the queries randomly and to present the worst and the most frequent execution results.

From the memory usage point of view, we have decided to divide the queries speed tests into two categories. In the first category (M1) will be the test results that do not use any cache, as a result, during the execution it is necessary to fetch all data from disk. The second category (M2) will contain the test results that may use cached items. After the query has been evaluated, the items are not removed from the cache and the evaluation of a similar query should be much faster. The M2 category may be considered as a partially in-memory query evaluation.

In order to find possible mistakes, we will also present the number of returned results. The size of the returned set usually has a big influence on the execution time and may help to understand the evaluation process. Of course we expect that the number of results returned by our index and Sesame will be equal.

### 6.3.1. Query 1

The first query simulates a general search. The customer specifies the product features of a known product type and expects to receive a list of products and their labels. Taking into account that it will access a big amount of data, the customer uses the ORDER BY, FILTER and LIMIT constraints.

```
# Query 1
SELECT DISTINCT ?product ?value1 ?label
WHERE {
  ?product rdfs:label "thudded nonirritating stenographic" .
  ?product a bsbm-inst:ProductType1.
  ?product bsbm:productFeature bsbm-inst:ProductFeature8781.
  ?product bsbm:productFeature bsbm-inst:ProductFeature976.
  ?product bsbm:productPropertyNumeric1 ?value1 .
  FILTER (?value1 > 400)
}
ORDER BY ?label
LIMIT 10
```

The query does not seem to be very complicated, but we would like to describe the way it is evaluated by the used SPARQL engine. At first it is necessary to say that the SPARQL engine does not provide us the full query and in order to evaluate a query it just tries to evaluate each clause (subquery). The evaluated clause also contains the information about the required number of solutions. For example, it may require the evaluation of the first clause, which contains one variable and initially a limit of 10 solutions. The library expects to receive the textual representation of the triple, which means, that we will have to access the FullDoc index. We believe that this is very inefficient because latter we have to evaluate the second clause, but this time, the clause will not contain any variables. The ?product variable will be bounded to the solutions from the first clause, as a result, we will have to check if the triples are valid. We may choose to create the hashes for each part of the triple and to search in the subject, predicate or object index, or we can create the hash of the full triple and we will search in the FullDoc index. Normally we will not choose the FullDoc index, because its items are big and will require multiple disk accesses.

If we could change the way SPARQL engine evaluates the clauses, the best would be to evaluate the first clause and just return the hash representations of the triple parts. The second clause will be evaluated by transforming the predicate and

object to their hash representation, after which we will just check, which solutions are valid. In the end we will retrieve the textual representation of the valid solutions.

Now that we have explained how queries with multiple clauses are evaluated, we will explain how the LIMIT constraint is evaluated. As it has already been said, the provided clauses contain the number of required solutions. This number cannot be smaller than the value from the LIMIT constraint. Nevertheless, it is necessary to take into consideration that not all solutions of the first clause are valid solutions of the second clause. As a result, the number of valid solutions for the entire query may be smaller than the specified number. The SPARQL engine solves this problem by evaluating the same query once again, but this time the required number of solutions is higher than the previous value. This process is repeated until the required number of final solutions is the same as the one required by the LIMIT constraint, or until the first clause gives the same number of solutions two times in a row. This kind of evaluation is probably not the most effective, but we cannot propose a better one, without complicating the entire evaluation process.

The next part that has to be analysed is the FILTER constraint. We have to notice that this part of the evaluation is entirely done by the SPARQL engine. This is one of the cases when it is necessary to retrieve the textual representation of the triple. The next case is the evaluation of the ORDER BY constraint.

After analysing this query we may conclude that the evaluation process may be divided into two parts. The first part operates with sets and performs the basic set operations such as union or intersection. The second part operates with the textual representation of the triple, by applying value constraints or sorting. Our implementation is capable to offer the necessary information in order to perform basic set operations and what is more important the index is designed to operate in this way. Nevertheless, we could not find a library that would be able to use it in the most efficient way. In order to demonstrate the power of our index, we have decided to implement in C# a static evaluation of some of the slowest queries. We will use the existing data structures and will try to evaluate the queries by operating with hashes.

From the following table we may notice that the in-memory evaluation M2 is faster than the Sesame index in both execution types. This may be explained by the fact that our index at first finds the subject that fits one of the required clauses and locates it in the subject index component. After this all the required information is nearly loaded (it depends from the size of the sub-tree that corresponds to the required subject) and all other clauses are evaluated in-memory.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	1	00:00.478	00:00.002	00:00.027
Frequent	1	00:00.005	00:00.001	00:00.002

Table 6 Query 1 results

### 6.3.2. Query 2

The next query is designed to access the information that is related to a known product. It simulates the situation when the consumer wants to view basic information about an already known product. This query is interesting because it returns more than 10 parameters in each result, which may be a real problem for indexes that do not keep the product parameters values in the same place. In our case it will be enough to search for the specified product in the subject index component, after which we will have the list or the sub-tree for the missing values (predicates and objects).

```
# Query 2
SELECT ?label ?comment ?producer ?productFeature
      ?propertyTextual1 ?propertyTextual2 ?propertyTextual3
      ?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4
      ?propertyTextual5 ?propertyNumeric4
WHERE {
  bsbm-inst-dfp:Product1 rdfs:label ?label .
  bsbm-inst-dfp:Product1 rdfs:comment ?comment .
  bsbm-inst-dfp:Product1 bsbm:producer ?p .
  ?p rdfs:label ?producer .
  bsbm-inst-dfp:Product1 dc:publisher ?p .
  bsbm-inst-dfp:Product1 bsbm:productFeature ?f .
  ?f rdfs:label ?productFeature .
  bsbm-inst-dfp:Product1 bsbm:productPropertyTextual1
?propertyTextual1 .
  bsbm-inst-dfp:Product1 bsbm:productPropertyTextual2
?propertyTextual2 .
  bsbm-inst-dfp:Product1 bsbm:productPropertyTextual3
?propertyTextual3 .
  bsbm-inst-dfp:Product1 bsbm:productPropertyNumeric1
?propertyNumeric1 .
  bsbm-inst-dfp:Product1 bsbm:productPropertyNumeric2
?propertyNumeric2 .
```



```

OPTIONAL { bsbm-inst-dfp:Product1 bsbm:productPropertyTextual4
?propertyTextual4 }
OPTIONAL { bsbm-inst-dfp:Product1 bsbm:productPropertyTextual5
?propertyTextual5 }
OPTIONAL { bsbm-inst-dfp:Product1 bsbm:productPropertyNumeric4
?propertyNumeric4 }
}

```

As we may see from the evaluation results that are presented in the following table, the difference between the M1 and M2 evaluation is approximately 0.020 milliseconds. The difference between our index (M2) and Sesame is very small and it is hard to say which one may be better.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	17	00:00.034	00:00.014	00:00.081
Frequent	17	00:00.032	00:00.008	00:00.007

Table 7 Query 2 results

### 6.3.3. Query 3

After viewing the product information, the customer may want to find another product with more specific parameters. This is the case when the query contains multiple FILTER constraints and what is more interesting it uses the negation of the BOUND constraint. The last is used to find the products that do not have a specified property.

```

# Query 3
SELECT ?product ?label ?testVar
WHERE {
  ?product rdfs:label ?label .
  ?product a bsbm-inst:ProductType1 .
  ?product bsbm:productFeature bsbm-inst:ProductFeature14 .
  ?product bsbm:productPropertyNumeric1 ?p1 .
  FILTER ( ?p1 > 400 )
  ?product bsbm:productPropertyNumeric3 ?p3 .
  FILTER ( ?p3 < 1000 )
  OPTIONAL {
    ?product bsbm:productFeature bsbm-inst:ProductFeature279 .
    ?product rdfs:label ?testVar }
  FILTER (!bound(?testVar))
}
ORDER BY ?label
LIMIT 10

```

The evaluation results are presented in the following table and once again the M1 evaluation type is slower than M2. Nevertheless, this time the speed of our index (M2) is approximately the same as of the Sesame. In order to evaluate the first

clause, our index uses the predicate index component, which returns a relatively big sub-tree. Just a few of the items from the sub-tree are valid solutions; however, the number of returned solutions is reduced by the LIMIT constraint. As a result it may be necessary to evaluate the first clause more than once. The fact that it has to evaluate the same query multiple times leads to long evaluation periods.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	10	00:01.774	0:00.651	00:02.197
Frequent	10	00:01.188	0:00.229	00:00.278

Table 8 Query 3 results

#### 6.3.4. Query 4

The following query is similar to the previous one, but in this case the customer wants to find all the products that have one of the two sets of features. For this purpose the query uses the UNION constraint.

```
# Query 4
SELECT DISTINCT ?product ?label ?propertyTextual
WHERE {
  {
    ?product rdfs:label ?label .
    ?product rdf:type bsbm-inst:ProductType1 .
    ?product bsbm:productFeature bsbm-inst:ProductFeature14 .
    ?product bsbm:productFeature bsbm-inst:ProductFeature10 .
    ?product bsbm:productPropertyTextual1 ?propertyTextual .
    ?product bsbm:productPropertyNumeric1 ?p1 .
    FILTER ( ?p1 > 400 )
  } UNION {
    ?product rdfs:label ?label .
    ?product rdf:type bsbm-inst:ProductType2 .
    ?product bsbm:productFeature bsbm-inst:ProductFeature14 .
    ?product bsbm:productFeature bsbm-inst:ProductFeature35 .
    ?product bsbm:productPropertyTextual1 ?propertyTextual .
    ?product bsbm:productPropertyNumeric2 ?p2 .
    FILTER ( ?p2 > 500 )
  }
}
ORDER BY ?label
OFFSET 5
LIMIT 10
```

If compared to the previous query, this one touches a much higher amount of data. Other two differences are the use of the OFFSET and UNION constraints. From the evaluation results that are presented in the following table we may say that this is one of the slowest queries. It is the perfect candidate for proving that if the SPARQL

engine had used our index in a correct way, it would have returned a valid solution in shorter time.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	10	05:29.007	00:18.750	00:02.281
Frequent	10	00:31.650	00:05.563	00:00.261

**Table 9 Query 4 results**

In order to show that the query may be evaluated in a more efficient way, we have implemented a few classes that permit us to specify the query and initialise the required program object instances without the need to parse the query. Another modification that we have performed is the removal of the ORDER BY, OFFSET and LIMIT constraints. Without these constraints, as it is presented in the following table, in the case of the M2 evaluation type, it took us 5.5 seconds to evaluate the query, while the SPARQL engine required 5.6 seconds. This may be explained by the fact that in both cases the required information is already loaded. In the case of the M1 evaluation type the difference between the static evaluation and the SPARQL engine is approximately 5 seconds. By showing this result we have proved that the efficiency of our index depends from the way it is used by the SPARQL engine.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	269	00:27.447	00:06.322	00:00.952
Frequent	269	00:22.240	00:05.624	00:00.232
Frequent (static)	269	00:17.140	00:05.531	00:00.232

**Table 10 Query 4 results (static evaluation)**

We could not improve the evaluation speed; nevertheless, we tried to find out what may be the problem and came to the conclusion that the slowest is the evaluation (M1) of the subqueries contained in the second UNION block. While the first UNION block provided all the 269 results, the second did not return any result, but took 6 seconds to evaluate. Another interesting observation is that after reordering the way each UNION block is evaluated, we managed to evaluate (M1) the query in lesser than 8 milliseconds.

### 6.3.5. Query 5

Now that the customer has found the products that fulfil his requirements, he may want to find the products that have similar features. For this he specifies the title of a known product and searches for products with a different title, but in the same time with similar features. If compared to previous queries, this one uses a more complex FILTER constraint.

```
# Query 5
SELECT DISTINCT ?product ?productLabel
WHERE {
  ?product rdfs:label ?productLabel .
  FILTER (bsbm-inst-dfp:Product2 != ?product)
  bsbm-inst-dfp:Product1 bsbm:productFeature ?prodFeature .
  ?product bsbm:productFeature ?prodFeature .
  bsbm-inst-dfp:Product1 bsbm:productPropertyNumeric1
?origProperty1 .
  ?product bsbm:productPropertyNumeric1 ?simProperty1 .
  FILTER (
    ?simProperty1 < (?origProperty1 + 120)
    && ?simProperty1 > (?origProperty1 - 120))
  bsbm-inst-dfp:Product1 bsbm:productPropertyNumeric2
?origProperty2 .
  ?product bsbm:productPropertyNumeric2 ?simProperty2 .
  FILTER (
    ?simProperty2 < (?origProperty2 + 170)
    && ?simProperty2 > (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	5	00:01.659	00:00.843	00:01.160
Frequent	5	00:01.610	00:00.676	00:00.722

Table 11 Query 5 results

If compared to the Sesame index, the in-memory evaluation (M1) is performed with approximately the same speed, while the disk evaluation needs as twice more time. In order to identify the valid solutions it is necessary to find all product features for “Product1”. This may be done by using the subject index component. Nonetheless, in order to find the value of the variable `?product`, it will be necessary to iterate through a big sub-tree from the predicate index component. Next it will be necessary to use once again the predicate index component, but this time it will be enough to iterate through a list of object keys that correspond to the known predicate and subject keys. After loading the sub-trees for the required values

of the `?product` variable it will be possible to evaluate the query in-memory. We may notice that the most complicated and time consuming operation is the iteration through the sub-tree while searching the values of the `?product` variable. The FILTER constraints are evaluated each time a clause returns a list of solutions. This is possible because the solutions are returned as textual representations of the RDF triples.

#### 6.3.6. Query 6

The 6<sup>th</sup> query is used to search the product by specifying only a part of the title, for this it is necessary to use the REGEX constraint.

```
# Query 6
SELECT ?product ?label
WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type bsbm:Product .
  FILTER regex(?label, "naut")
}
```

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	17	01:14.330	00:09.624	00:12.426
Frequent	17	00:32.361	00:05.134	00:02.290

Table 12 Query 6 results

These may seem to be one of the cases when it is not possible to use the power of the hashed triples, but we believe that the first two clauses may be evaluated by searching there subject key sets intersection. After minimising the solution set, we will fetch the textual representation of the triple and evaluate the REGEX constraint. After implementing the static evaluation, we have managed to minimise the required evaluation time, so the speed of the in-memory evaluation is just 0.5 milliseconds slower than the Sesame index.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst (static)	17	00:16.614	00:08.427	00:12.426
Frequent (static)	17	00:15.756	00:02.729	00:02.290

Table 13 Query 6 results (static evaluation)

### 6.3.7. Query 7

The following query is used to offer detailed information about a product, including the list of offers from the vendors, reviews and ratings if any exist. The last two features are optional, but the review may be represented by a longer text.

```
# Query 7
SELECT ?productLabel ?offer ?price
      ?vendor ?vendorTitle ?review ?revTitle
      ?reviewer ?revName ?rating1 ?rating2
WHERE {
  bsbm-inst-dfp:Product2 rdfs:label ?productLabel .
  OPTIONAL {
    ?offer bsbm:product bsbm-inst-dfp:Product2 .
    ?offer bsbm:price ?price .
    ?offer bsbm:vendor ?vendor .
    ?vendor rdfs:label ?vendorTitle .
    ?vendor bsbm:country <http://download.org/rdf/iso-
3166/countries#DE> .
    ?offer dc:publisher ?vendor .
    ?offer bsbm:validTo ?date .
    FILTER (?date > 2000-07-01 )
  }
  OPTIONAL {
    ?review bsbm:reviewFor bsbm-inst-dfp:Product2 .
    ?review rev:reviewer ?reviewer .
    ?reviewer foaf:name ?revName .
    ?review dc:title ?revTitle .
    OPTIONAL { ?review bsbm:rating1 ?rating1 . }
    OPTIONAL { ?review bsbm:rating2 ?rating2 . }
  }
}
```

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	2	00:00.941	00:00.013	00:00.163
Frequent	2	00:00.023	00:00.004	00:00.005

Table 14 Query 7 results

The query evaluation is not very complicated and as the results table shows, the speed of the M2 evaluation type and Sesame are equal.

### 6.3.8. Query 8

The next query just searches the most recent 20 reviews for a product, which are written in English language. Unfortunately the used SPARQL engine does not use the provided language property, and as a result returns all the reviews, without filtering by the required language. We have tried to find the reason but with no success.

```

# Query 8
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1 ?rating2 ?rating3 ?rating4
WHERE {
  ?review bsbm:reviewFor bsbm-inst-dfp:Product2 .
  ?review dc:title ?title .
  ?review rev:text ?text .
  FILTER langMatches( lang(?text), "en" )
  ?review bsbm:reviewDate ?reviewDate .
  ?review rev:reviewer ?reviewer .
  ?reviewer foaf:name ?reviewerName .
  OPTIONAL { ?review bsbm:rating1 ?rating1 . }
  OPTIONAL { ?review bsbm:rating2 ?rating2 . }
  OPTIONAL { ?review bsbm:rating3 ?rating3 . }
  OPTIONAL { ?review bsbm:rating4 ?rating4 . }
}
ORDER BY DESC(?reviewDate)
LIMIT 20

```

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	2	00:00.216	00:00.009	00:00.186
Frequent	2	00:00.016	00:00.003	00:00.003

Table 15 Query 8 results

### 6.3.9. Query 9

In order to decide whether to trust a review, the consumer asks for any kind of information about the reviewer. This is the case when in place of SELECT we use the DESCRIBE query form.

```

# Query 9
DESCRIBE ?x
WHERE { dataFromRatingSite1:Review3 rev:reviewer ?x }

```

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	30	00:00.114	00:00.021	01:04.282
Frequent	30	00:00.057	00:00.009	00:02.912

Table 16 Query 9 results

We were surprised by the results of this query, mainly because of the high difference in the evaluation speed of our index and Sesame. We have tried to run the query multiple times and have compared the results statement by statement, but we could not find any difference. This probably means that the Sesame index is not optimised for such queries. In order to find the solutions, our query at first returns the values of the ?x variable, this may be done very fast just by searching the subject or predicate index component. Next it is necessary to find all the ?x variable values in

the object index component and fetch the corresponding triples. Once again this is done very fast because the sub-tree that corresponds to the specified object key, contains all the required triple keys. The final result is returned after fetching the data from the FullDoc index component.

### 6.3.10. Query 10

Next the customer decides to buy the product from an USA vendor that may deliver the product within 3 days. Of course he wants to select the cheapest offer; this is why he sorts the results by the price value. This query also demonstrates the use of the date data type. The last FILTER constraint contains the data type specification `xsd:dateTime`.

```
# Query 10
SELECT DISTINCT ?offer ?price ?date
WHERE {
  ?offer bsbm:product bsbm-inst-dfp:Product1 .
  ?offer bsbm:vendor ?vendor .
  ?offer dc:publisher ?vendor .
  ?vendor bsbm:country <http://download.org/rdf/iso-3166/countries#US> .
  ?offer bsbm:deliveryDays ?deliveryDays .
  FILTER (?deliveryDays <= 3)
  ?offer bsbm:price ?price .
  ?offer bsbm:validTo ?date .
  FILTER (?date >= "2008-07-01T00:00:00"^^xsd:dateTime )
}
ORDER BY xsd:double(str(?price))
LIMIT 10
```

Once again the M1 evaluation type of our index is a little faster than the Sesame index. This time the LIMIT constraint had no influence on the evaluation process, because the first clause returned just 7 triples, which could be evaluated as valid solutions. The rest of the evaluation is done using the predicate index component.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	1	00:00.245	00:00.009	00:00.012
Frequent	1	00:00.019	00:00.004	00:00.009

Table 17 Query 10 results



### 6.3.11. Query 11

After selecting the best offer the customer may want to read all the existing information about the offer and the vendor.

```
# Query 11
SELECT ?property ?hasValue ?isValueOf
WHERE {
  { dataFromVendor1:Offer1 ?property ?hasValue }
  UNION
  { ?isValueOf ?property dataFromVendor1:Offer1 }
}
```

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	10	00:00.011	00:00.004	00:17.140
Frequent	10	00:00.006	00:00.001	00:02.884

Table 18 Query 11 results

We have checked if the returned results are the same, and it seems that our index in this precise case is much faster than the Sesame index. In order to evaluate this query our index just fetches all the triples that correspond to the first and second clauses, and the triples with the same predicate are returned. In order to do it, it is necessary to iterate through the subject and object sub-trees that correspond to the specified subject and object value. The size of these two sub-trees is small as a result the evaluation is fast. It is hard to say why the Sesame index is slower. May be it evaluates the first clause and the second clause is evaluated in correspondence to the returned predicate values.

### 6.3.12. Query 12

In the end the customer may want to save the offer information on his own computer, using an RDF scheme that he will define by himself.

```
# Query 12
CONSTRUCT { dataFromVendor1:Offer1 bsbm-export:product ?productURI .
  dataFromVendor1:Offer1 bsbm-export:productlabel ?productlabel .
  dataFromVendor1:Offer1 bsbm-export:vendor ?vendorname .
  dataFromVendor1:Offer1 bsbm-export:vendorhomepage
?vendorhomepage .
  dataFromVendor1:Offer1 bsbm-export:offerURL ?offerURL .
  dataFromVendor1:Offer1 bsbm-export:price ?price .
  dataFromVendor1:Offer1 bsbm-export:deliveryDays ?deliveryDays .
  dataFromVendor1:Offer1 bsbm-export:validuntil ?validTo }
WHERE { dataFromVendor1:Offer1 bsbm:product ?productURI .
  ?productURI rdfs:label ?productlabel .
```

```

dataFromVendor1:Offer1 bsbm:vendor ?vendorURI .
?vendorURI rdfs:label ?vendorname .
?vendorURI foaf:homepage ?vendorhomepage .
dataFromVendor1:Offer1 bsbm:offerWebpage ?offerURL .
dataFromVendor1:Offer1 bsbm:price ?price .
dataFromVendor1:Offer1 bsbm:deliveryDays ?deliveryDays .
dataFromVendor1:Offer1 bsbm:validTo ?validTo }

```

In order to evaluate the above query, our index uses the subject index component. As a result, after the execution of the first clause, a big part of the required information is already loaded.

Execution type	Results	Our index (M1)	Our index (M2)	Sesame
Worst	8	00:00.014	00:00.005	00:00.014
Frequent	8	00:00.009	00:00.002	00:00.008

Table 19 Query 12 results

## 6.4. Conclusion

The experiments had to prove that our implementation is able to index big RDF files without losing any triples and that it may offer SPARQL query responses comparable to existing RDF indexes. By indexing the dataset that contained 25 millions of triples we have tested the maximum effectiveness of the indexation process. The indexation of the 35 millions of triples dataset could not be completed, because of a failure during the creation of the object index component. Unfortunately the indexation speed and as a result effectiveness is not as good as we have expected. The indexation speed may be different on some PCs even if there configuration is more or lesser the same. On the other hand, it is important that we were able to index RDF files that could not be indexed by other two projects. We suppose that those projects are more effective on more powerful computers, while we wanted to create an index that would be able to work on computers with a less powerful configuration (simple PC).

The second part of the experiment has proved that the SPARQL query evaluation time depends from the used SPARQL engine. First of all, the chosen SPARQL engine is not designed to use the hashed values and to operate with sets of hashed values. Also we have to mention that the way it evaluates the LIMIT constraint is not effective enough. A much better approach is to be able to iterate

through the solutions of each subquery and eventually to build the result set of the required size. The problem is that the SPARQL engine does not offer the possibility to implement such logic.

The next important observation is the fact that the query evaluation speed depends from the order in which subqueries are evaluated. By evaluating the subqueries of 4<sup>th</sup> and 6<sup>th</sup> query in a different order, we have decreased the evaluation period by 2-3 times. In order to offer such functionality we would probably have to add to each node item the information about the number of child items that it has. As a result the SPARQL engine would be able to optimise the order in which the queries have to be evaluated. Unfortunately the SPARQL engine that we are using does not support this.

The SPARQL evaluation test results for 1000 random evaluations of the BSBM queries may be found in the “Sparql-Results.xslt” file. The Excel file contains the data that have been extracted from the “dataset\_result.txt” file.

## 7. Concluding remarks

In the first chapter we have stated our goals: to be able to index files bigger than 2 GB, to create an RDF index that will be able to compete with existing indexes and to explain in detail our solution.

As the test results have showed, we successfully indexed a dataset that contained 25 millions of triples and occupied approximately 3 GB on disk. The required time to complete this task was not as good as we initially expected, but none of the projects, which are considered to be some of the best server solutions and with which we have compared our project, did manage to complete the task. We believe that the creation of a second level sub-tree, as it has been described in the indexation testing section, will result in a higher indexation speed and the possibility to index bigger datasets. Nonetheless, the structure that we use in order to create the first level sub-tree is not general enough to support the creation of the second level sub-tree.

We have managed to compare the evaluation speed of the SPARQL queries with the Sesame project. It is true that some queries have required more time in order to be evaluated, but on the other hand many of the queries have been evaluated with approximately the same or even a higher speed. In the testing chapter we have mentioned that the lack of an adequate SPARQL engine is an important issue, but unfortunately we did not have enough resources to solve it. We believe that this issue may be solved in some future project and with no doubt it is a success that the index was capable to answer to all SPARQL queries in reasonable time.

The last but not the least important goal for us was to explain in detail our solution. We did not present any source code or classes because we believe that it would make the understanding much harder. We have oriented our explanation onto the description of the steps and the decisions that we have made during the design and implementation phase. We also consider that the testing chapter plays a big role in understanding our solution. By describing the graphs and the results of the SPARQL query evaluation we have revealed the weak and strong points of our index.

## Bibliography

1. Linked data. [Online] [http://en.wikipedia.org/Linked\\_data](http://en.wikipedia.org/Linked_data).
2. Linked data. [Online] [http://structureddynamics.com/linked\\_data.html](http://structureddynamics.com/linked_data.html).
3. Ontology. [Online] 2011.  
[http://en.wikipedia.org/wiki/Ontology\\_%28information\\_science%29](http://en.wikipedia.org/wiki/Ontology_%28information_science%29).
4. Linked data. [Online] <http://www.w3.org/DesignIssues/LinkedData>.
5. Linked Data vs Open Data vs RDF Data. [Online] 2011.  
<http://blogs.ecs.soton.ac.uk/webteam/2011/07/17/linked-data-vs-open-data-vs-rdf-data/>.
6. **Harth, Andreas and Harth, Andreas.** Yet another RDF store: Perfect index structures for storing semantic web data with contexts. [Online]  
[http://www.uniportal.com.br/media/marcel/download\\_gallery/+++www2005-submission.pdf](http://www.uniportal.com.br/media/marcel/download_gallery/+++www2005-submission.pdf).
7. Redland RDF Storage and Retrieval. [Online]  
<http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/bristol-1.html>.
8. **David, Beckett.** Redland RDF Application Framework. [Online] 2005.
9. **Harris, Stephen and Gibbins, Nicholas.** 3store: Efficient Bulk RDF Storage. [Online]  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.1011&rep=rep1&type=pdf>.
10. **Luaces, Miguel R., et al.** An Ontology-Based Index to Retrieve Documents with Geographic Information. [Online] 2008.  
<http://www.springerlink.com/content/b12703252g67124q/fulltext.pdf>.
11. **Wilkinson, Kevin, et al.** Efficient RDF Storage and Retrieval in Jena2. [Online]  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.126.5486&rep=rep1&type=pdf>.
12. The Jena SemanticWeb Toolkit. [Online]  
<http://www.hpl.hp.com/semweb/jena.htm>.
13. Jena2 Database Interface. [Online]  
<http://jena.sourceforge.net/DB/index.html>.

14. Jena2 Database Interface - Database Layout. [Online]  
<http://jena.sourceforge.net/DB/layout.html>.
15. TDB Architecture. [Online]  
<http://incubator.apache.org/jena/documentation/tdb/architecture.html>.
16. SPARQL query processing with conventional relational database systems.  
[Online] <http://eprints.ecs.soton.ac.uk/11126/1/harris-ssws05.pdf>.
17. Scalability report on triple store applications. [Online] 26 07 2004.  
<http://simile.mit.edu/reports/stores/>.
18. Scalable storage solution for next generation knowledge services.  
[Online] <http://www.aktors.org/technologies/3store/>.
19. Sesame project description. [Online] <http://www.openrdf.org/>.
20. **Broekstra, Jeen and Kampman, Arjohn.** Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. [Online]  
<http://www.few.vu.nl/~frankh/postscript/MIT01.pdf>.
21. Sesame framework. [Online]  
[http://en.wikipedia.org/wiki/Sesame\\_%28framework%29](http://en.wikipedia.org/wiki/Sesame_%28framework%29).
22. Native store configuration. [Online]  
<http://www.openrdf.org/doc/sesame2/users/ch07.html#section-native-store-config>.
23. SPARQL Query Language for RDF. [Online] 15 1 2008.  
<http://www.w3.org/TR/rdf-sparql-query/>.
24. SPARQL. [Online] <http://en.wikipedia.org/wiki/SPARQL>.
25. SPARQL update. [Online] 15 7 2008.  
<http://www.w3.org/Submission/SPARQL-Update/>.
26. Search RDF data with SPARQL. [Online] 10 5 2005.  
<http://www.ibm.com/developerworks/xml/library/j-sparql/>.
27. Turtle - Terse RDF Triple Language. [Online] 28 3 2011.  
<http://www.w3.org/TeamSubmission/turtle>.
28. **Mohamed, K. A.** Lecture Theme 03 – Part I. *Advanced Algorithms & Data Structures*.
29. **Guttman, Antonin.** R-trees a dynamic index structure for spatial searching. [Online] <http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf>.
30. **Peter Scheuermann, Mohamed Ouksel.** Multidimensional B-trees for associative searching in database systems. [Online] 1981. <http://ac.els->

cdn.com/0306437982900242/1-s2.0-0306437982900242-main.pdf?\_tid=768ee164e1854eeb6b33f2011e619a45&acdnat=1333053541\_69b133b2cae1bb7c21b172634141bfa5.

31. **Leslie, Harry, et al.** Efficient search of multidimensional B-trees. [Online] <http://www.vldb.org/conf/1995/P710.PDF>.
32. BPlusTree. [Online] <http://csharptest.net/projects/bplustree/>.
33. **Lehman, Philip L. and Yao, S Bing.** Efficient locking for concurrent operations on B-Trees. [Online] <http://www.cs.cornell.edu/courses/cs4411/2009sp/blink.pdf>.
34. Berlin SPARQL Benchmark (BSBM). [Online] <http://www4.wiwiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/>.
35. LUBM: A Benchmark for OWL Knowledge Base Systems. [Online] <http://swat.cse.lehigh.edu/projects/lubm/>.
36. Social Network Intelligence BenchMark. [Online] [http://www.w3.org/wiki/Social\\_Network\\_Intelligence\\_BenchMark](http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark).
37. **Morsey, Mohamed, et al.** DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. [Online] [http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research\\_Paper/03/70310448.pdf](http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research_Paper/03/70310448.pdf).
38. **Jiawei Han, Micheline Kamber.** *Data mining: concepts and techniques (3-d edition)*. 2012.
39. **Roychowdhury, Ruku and Kariwala, Shagun.** Report On AllegroGraph RDFStore. <http://tinman.cs.gsu.edu>. [Online] 2011. <http://tinman.cs.gsu.edu/~raj/8711/sp11/presentations/allegroGraphReport.pdf>.
40. **Minack, Enrico, et al.** The Sesame LuceneSail: RDF Queries with Full-text Search. [Online] <http://www.dfki.uni-kl.de/~sauermann/papers/Minack%2B2008.pdf>.
41. **Harth, Andreas and Decker, Stefan.** Optimized Index Structures for Querying RDF from the Web. [Online] <http://dip.semanticweb.org/documents/Harth-Decker-yars.pdf>.
42. **Pugliese, Andrea and Udre, Octavian.** Scaling RDF with Time. [Online] <http://www.conference.org/www2008/papers/pdf/p605-puglieseA.pdf>.

43. **Mohamed, K. A.** *R-Tree: Spatial Representation on a Dynamic-Index Structure*. 2006.
44. **Levandowski, Justin J. and Mokbel, Mohamed F.** RDF Data-Centric Storage. [Online] <http://www-users.cs.umn.edu/~mokbel/papers/icws09.pdf>.
45. Bigdata® Scale-out Architecture (Draft). [Online] <http://sourceforge.net/projects/bigdata/>.
46. Comparison of Triple Stores. [Online] [http://www.bioontology.org/wiki/images/6/6a/Triple\\_Stores.pdf](http://www.bioontology.org/wiki/images/6/6a/Triple_Stores.pdf).
47. **Abadi, Daniel J., et al.** Scalable Semantic Web Data Management Using Vertical Partitioning. [Online] <http://cs-www.cs.yale.edu/homes/dna/abadirdf.pdf>.
48. **Freeston, Michael.** A general solution of the n-dimensionanl B-tree problem. [Online] <http://dl.acm.org/citation.cfm?id=568271.223796>.
49. **He, Huahai and Singh, Ambuj K.** Closure-Tree: An Index Structure for Graph Queries. [Online] <http://www.computer.org/portal/web/cSDL/doi/10.1109/ICDE.2006.37>.
50. **Knechtel, Martin, Hreno, Jan and Furdik, Karol.** Analysis of Semantic Stores and Specific ebbits Use Cases. [Online] [http://www.ebbits-project.eu/downloads/deliverables/D4.1\\_Analysis\\_of\\_Semantic\\_Stores\\_and\\_Specific\\_ebbits\\_Use\\_Cases.pdf](http://www.ebbits-project.eu/downloads/deliverables/D4.1_Analysis_of_Semantic_Stores_and_Specific_ebbits_Use_Cases.pdf).
51. **Pettovello, P. Mark and Fotouhi, Farshad.** MTree: An XML XPath Graph Index. [Online] <http://dl.acm.org/citation.cfm?id=1141389>.
52. **Udrea, Octavian, Pugliese, Andrea and Subrahmanian, V.S.** GRIN: A Graph Based RDF Index. [Online] <http://www.aaai.org/Papers/AAAI/2007/AAAI07-232.pdf>.
53. **Weiske, Christian and Auer, Sören.** Implementing SPARQL Support for Relational Databases. [Online] <http://cweiske.de/download/misc/SPARQL%20to%20SQL%20rewriting%20and%20enhancements.pdf>.
54. **Yuan, Dayu and Mitra, Prasenjit.** A Lattice-based Graph Index for Subgraph Search. [Online] <http://webdb2011.rutgers.edu/papers/Paper%2025/main.pdf>.



55. **Zou, Lei, et al.** Summarization Graph Indexing: Beyond Frequent Structure-Based Approach. [Online]  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.193.5786>.
56. **Oren, Eyal, Stenzhorn, Holger and Tummarello, Giovanni.** Sindice.com: a document-oriented lookup inde for open linked data. [Online] 2008.  
<http://www.eyaloren.org/pubs/ijmso2008.pdf>.
57. **Tran, Thanh, Ladwig, Gunter and Rudolph, Sebastian.** iStore: Efficient RDF data management using structure indexes for general grph structured data. [Online] <http://people.aifb.kit.edu/dtr/papers/strucidxTR.pdf>.
58. **Bizer, Christian, Heath, Tom and Berners-Lee, Tim.** Linked Data - The Story So Far. [Online] <http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf>.

## List of figures

Figure 1 Redland list of used hashes .....	21
Figure 2 Jena1 RDMBS Schema .....	23
Figure 3 Jena2 RDBMS Schema .....	24
Figure 4 3Store RDBMS Schema .....	28
Figure 5 Basic graph structure for a single contributor in bloggers.rdf [26] .....	33
Figure 6 Schematic query evaluation example .....	34
Figure 7 R-tree binding region .....	40
Figure 8 R-tree structure example.....	41
Figure 9 Schematic representation of a multidimensional B-Tree.....	43
Figure 10 MDBT Partial match example .....	44
Figure 11 Simple block structure .....	50
Figure 12 Smart block structure .....	51
Figure 13 Single smart block item information .....	52
Figure 14 B+-tree node structure .....	54
Figure 15 Adding a new empty block .....	55
Figure 16 Layered cache insert .....	57
Figure 17 LRU structure inserting a new item.....	63
Figure 18 Berlin benchmark dataset structure .....	68
Figure 19 RDF indexation window .....	110
Figure 20 Hash window .....	111
Figure 21 SPARQL window .....	112
Figure 22 Display index window .....	112
Figure 23 Settings window.....	113
Figure 24 Settings window (advanced settings).....	114
Figure 25 Statistics window .....	114

## List of graphs

Graph 1 Indexation speed PC1 (1 million)	70
Graph 2 Indexation speed PC2 (1 million)	71
Graph 3 Writing variable size blocks to cache PC1 (1 million)	73
Graph 4 Writing variable size blocks to cache PC2 (1 million)	73
Graph 5 Reading variable size blocks from cache PC1 (1 million)	74
Graph 6 Reading variable size blocks from cache PC2 (1 million)	74
Graph 7 Persistence effectiveness PC1 (1 million)	75
Graph 8 Persistence effectiveness PC2 (1 million)	75
Graph 9 Indexation speed PC1 (10 millions)	77
Graph 10 Indexation speed PC2 (10 millions)	77
Graph 11 Writing variable size blocks to cache PC1 (10 millions)	78
Graph 12 Writing variable size blocks to cache PC2 (10 millions)	78
Graph 13 Writing variable size blocks to disk PC1 (10 millions)	79
Graph 14 Writing variable size blocks to disk PC2 (10 millions)	79
Graph 15 Reading variable size blocks from cache PC1 (10 millions)	80
Graph 16 Reading variable size blocks from cache PC2 (10 millions)	80
Graph 17 Persistence effectiveness PC1 (10 millions)	81
Graph 18 Persistence effectiveness PC2 (10 millions)	81

## List of tables

Table 1 YARS list of indexes [6]	26
Table 2 Benchmark namespaces	68
Table 3 Indexation speed PC1	83
Table 4 Indexation speed PC2	83
Table 5 Index size	84
Table 6 Query 1 results	88
Table 7 Query 2 results	89
Table 8 Query 3 results	90
Table 9 Query 4 results	91
Table 10 Query 4 results (static evaluation)	91
Table 11 Query 5 results	92
Table 12 Query 6 results	93
Table 13 Query 6 results (static evaluation)	93
Table 14 Query 7 results	94
Table 15 Query 8 results	95
Table 16 Query 9 results	95
Table 17 Query 10 results	96
Table 18 Query 11 results	97
Table 19 Query 12 results	98

## Appendix A – User documentation

### User interface

The user interface offers just a few functions, but they should be enough in order to: build an index, inspect it or run SPARQL queries on it. The vast majority of the user interface components have been used for testing purposes during the implementation phase. It requires the Windows .Net 3.5 environment.

### RDF indexation window

The main window “RDF indexation”, which screenshot is presented further, offers the information about the current state of the program. It is composed from three parts: the menu, the information panel and the status bar. From the presented screenshot we may see the four menu items:

- File – the file menu contains only one option “Select dataset”. This is used to select the dataset that has to be indexed. The information about the selected dataset will be presented in the information panel.
- Index – this entry contains four items: “Start”, “Pause”, “Resume”, “Stop” and “Statistics”. The “Start” option lunches the indexation process. The “Pause” option freezes the reading of data from the dataset; as a result the indexation is paused as well. The “Resume” option unfreezes the reading of data from the dataset; as a result the indexation is resumed as well. The “Stop” option cancels the reading of data from the dataset (without the possibility to resume the process) and closes all resources that are used by the indexation process. The “Statistics” option initialises the “Statistics” window that displays the information about the indexation process in a graphic mode.
- Development – this entry contains three items: “Hash”, “SPARQL” and “Display index”. Each of these items initialises a new window. The “Hash” option initialises the “Hash” window that is used for the transformation of the inputted text to its hash value. The “SPARQL” option initialises the “SPARQL query” window that is used for the evaluation of the SPARQL queries. The “Display

index” option initialises the “Display index” window that offers the possibility to navigate through the index.

- Settings – this entry initialises the “Settings” window through which it is possible to change some of the indexation process settings.

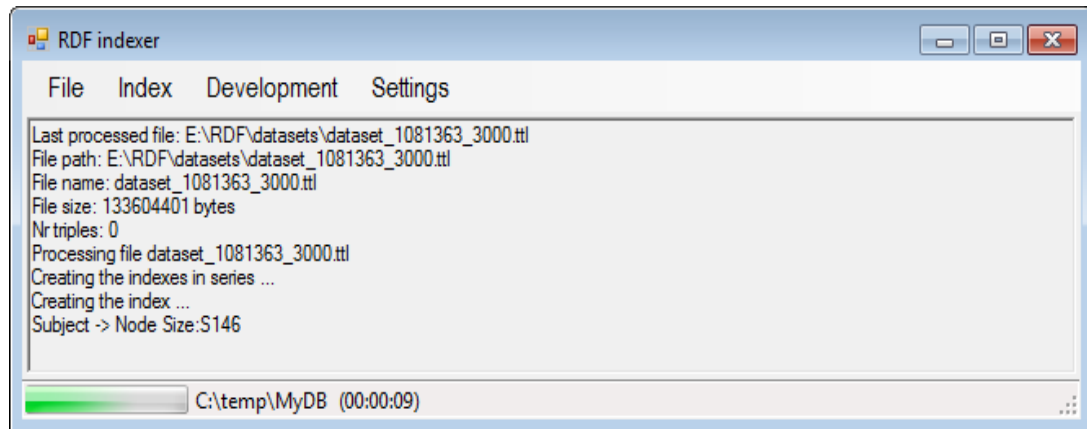


Figure 19 RDF indexation window

In order to make the process easier and faster, the program stores the last used index and the last used dataset. The information about these two parameters is presented in the status bar (in the above screenshot it is equal to “c:\temp\MyDb”) and respectively in the information panel (in the above image it is equal to “Last processed file ...”).

The next important part of the “RDF indexation” window is the information panel. We have already mentioned some of the information that it offers. From the above image we may notice that it also contains the information about the selected dataset path, file name and size. Initially we wanted to display the number of triples, but this may take a few minutes to compute, as a result it is turned off. According to the provided settings the information panel displays the information about the indexation process. This normally includes the title of the index component that is being built or the time in seconds that was required in order to build the index.

The last part is the status bar that contains a progress bar and a timer. The progress bar is used just to inform that the indexation process is running, while the timer counts the time that has passed from the moment the indexation has started.

## Hash window

The “Hash” window is presented in the following screenshot. It is just used for the transformation of the inputted text to the outputted hashed value.

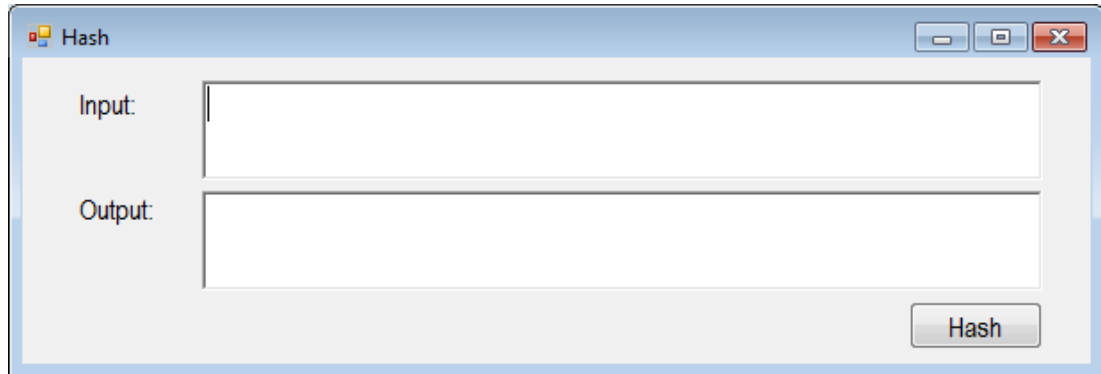


Figure 20 Hash window

## SPARQL window

The “SPARQL” window is used for SPARQL query evaluation. This window is divided into three parts: the SPARQL query panel, the SPARQL query result panel, the list of actions and the status bar. The SPARQL query panel is editable and has to contain the query that has to be evaluated. The SPARQL query result panel contains the result of the query that has been evaluated. Besides the query evaluation it is possible to save the query into a plain text file and latter to load it. In order to load an existing plain text file, which contains the query, it is required to select it from the list of offered file names (the combo box). This list is populated with the files that are located in the folder that is specified in the settings section. In order to save the query into a new plain text file it is necessary to click the “Save as” button. The “Save” button will save the query in the currently selected file. The “Delete” button will permanently delete the selected file. The “Evaluate” button starts the SPRQL query evaluation. During the query evaluation a progress bar in the status bar indicates the fact that the evaluation is still running. The “Random” button starts 100 random queries in series, from the list of queries. The query evaluation results are automatically saved into the “dataset\_results.txt” file that is contained in the “Results” folder, from the path that is specified in the settings.

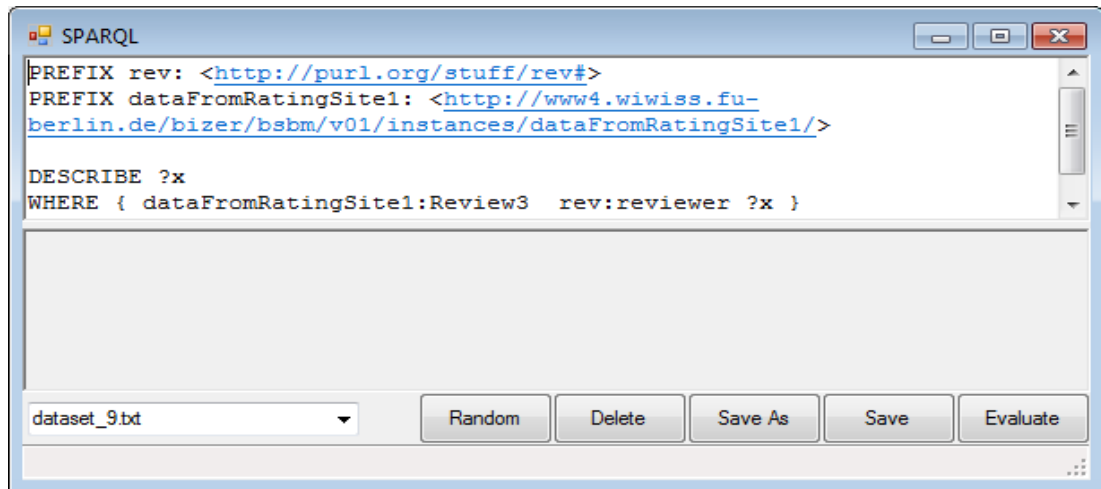


Figure 21 SPARQL window

## Display index window

The “Display index” window offers the possibility to visualise the index. In order to initialise the indexes root it is necessary to select the “Display tree” option from the upper menu, after which the left panel will contain four items, one for each index components. The component child items are displayed after double clicking on the desired parent item. If the item will be a leaf, then a window with the detailed information will popup. Each item contains information about the internal node: the key, the smart block address and the number of child items.

Another useful function is the possibility to search for the nodes that have the specified key. In order to do so, it is necessary to input the hashed value and to select the index component that has to be used. The results are displayed in a table on the right side of the window. It is important to remember that the provided hashed value has to be in the main B+-tree.

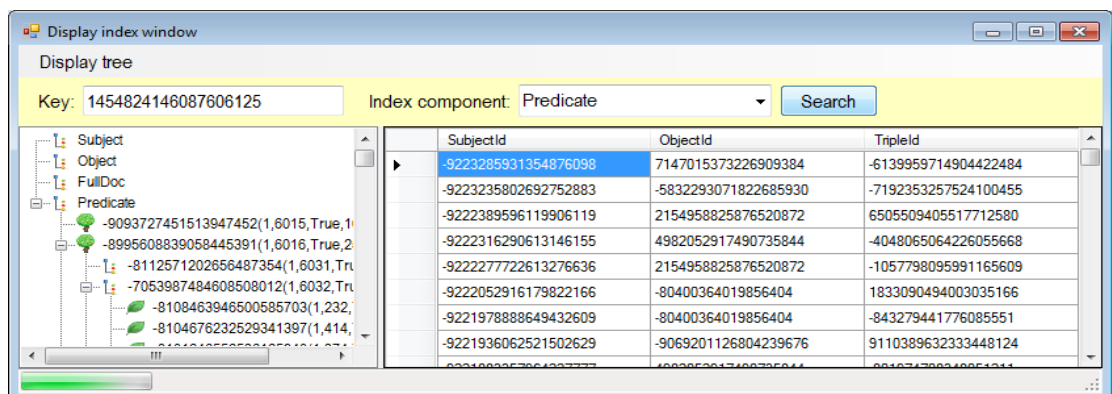


Figure 22 Display index window



## Settings window

The next important window is the “Settings” window. It contains a variety of parameters that can be changed. In this way it is possible to modify the index components structure and the indexation process. First of all it offers the possibility to set which index components have to be constructed and the indexation method: in series or in parallel. Next it is possible to set the location where the index will be built or the location of an existing index.

By clicking on the “Advanced” button the advanced settings become visible. These include: the number of items in a node, the number of dictionaries in the cache and their size for each type of block. Unfortunately it is not possible to define the leaves type.

The last but not the least important functions are provided by the “Export” and “Import” buttons. After clicking the “Export” button all the settings are exported to a plain text file called “settings.txt” that is located in the same directory as the index itself. The “Import” button reads the contents of the “settings.txt” file and according to these values changes the parameters values from the “Settings” window.

The “Default” button sets the parameters values to the values that have been evaluated as optimal, during the testing phase.

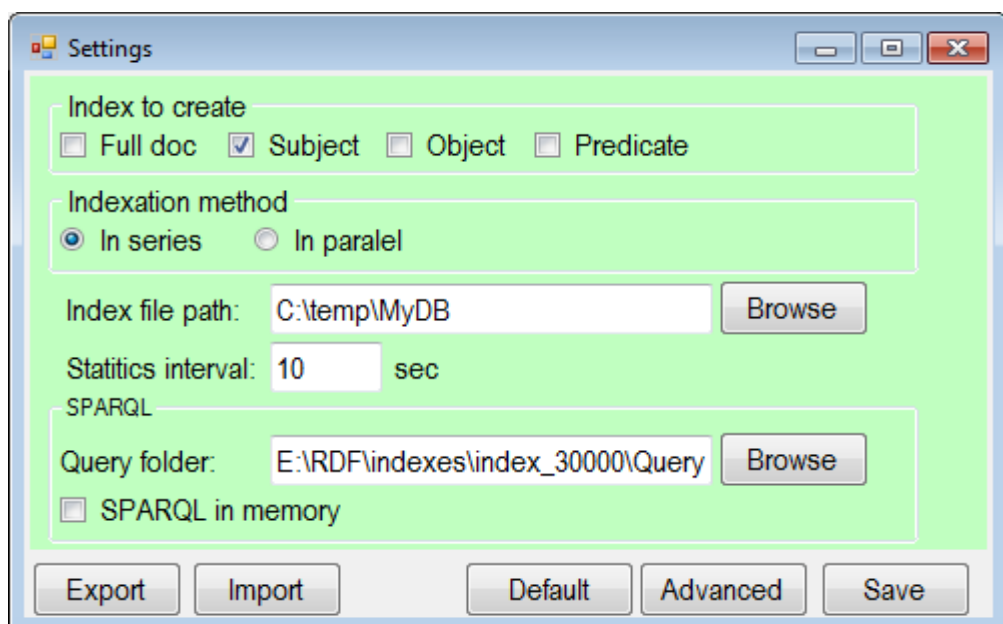


Figure 23 Settings window

Figure 24 Settings window (advanced settings)

## Statistics window

The “Statistics” window displays during the indexation process the values of some of the parameters that have been analysed during the optimisation and testing phase. The statistics are divided into four tabs, one for each index component. By checking a parameter it is possible to display or hide the line in the graph that corresponds to the modified parameter. It is not recommended to use it, because it decreases the indexation speed. Also by using the “Settings” window, it is possible to set the time interval in which the statistics will be generated. A short time interval may slow the indexation process, because it is necessary to write more than 30 parameters to a file on disk. The statistics have been described in the testing chapter.

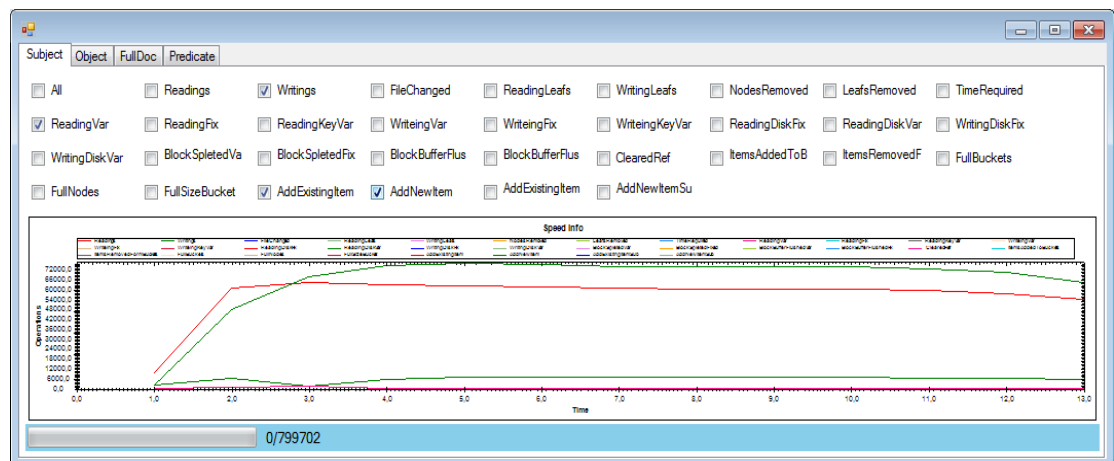


Figure 25 Statistics window

## Appendix B – Content of the attached DVD-ROM

The thesis is accompanied by a DVD ROM containing the source codes, compiled indexer and necessary files in order to run the tests. The structure of the DVD ROM is as follows:

/Berlin Benchmark

Contains the Berlin benchmark dataset generator and the SPARQL queries.

/Bin

Contains the binary files that correspond to the project that we have created.

/Datasets

Contains the archives of the datasets that have been used during the testing phase.

/Doc

Contains the diploma theses in the PDF format.

/Failed projects

Contains the “dotNetRDF” and “Jena” projects that have failed to complete the indexation and have been mentioned in the diploma thesis.

/Indexation results

Contains the Excel files with the statistics that have been created during the indexation.

/Indexes

Contains the indexes that have been created for the 1 and 10 millions triple datasets.

/RDFIndex

Contains the full project of the C# index implementation prototype. The project has been created in Microsoft Visual Studio 2008.

/Sesame

Contains the Sesame library and the Java project that we have created in order to run the indexation and the tests using the Sesame.

/SPARQL results

Contains the results that have been generated during the random evaluation of the Berlin benchmark queries.

## Appendix C – Program settings

The program offers the following types of settings:

### 1. Basic

**Index to create** – specifies that during the indexation process it is necessary to create the selected index components.

**Indexation method** – specifies that during the indexation it is necessary to create the specified index components “in parallel” or “in series”. During the “in series” method the index components are created individually, only one index component at a time. During the “in parallel” method the index components are created simultaneously.

**Index file path** – the directory that will contain or already contains the index files.

**Statistics interval** – the time interval (in seconds) after which it is necessary to write the indexation statistics to the appropriate file. A very short time interval may slow down the indexation process.

**Query folder** – the directory that contains the list of file with SPARQL queries.

**SPARQL in memory** – has to be checked if after the evaluation of a SPARQL query the fetched nodes may remain in main memory.

### 2. Advanced

**Internal nodes Max size** – the maximum size (in bytes) that the internal nodes from all B+-tree may occupy. If the size of the internal nodes is bigger than a part of the nodes is persisted and destroyed.

For each index component it is possible to modify the following parameters:

**Base node size** – the minimum number of items each internal node has to contain, from the main B+-tree (first tree level) (except the root node).

**Internal node size** – the minimum number of items in each internal node has to contain, from the sub-trees (second tree level) (except the root node). The FullDoc index component does not support this parameter.

Blocks are divided into two types: fix and variable (var) size blocks. For each block type it is possible to set the following parameters:

**Number of dictionaries** – the number of dictionaries (see definition in main document) that are used in the cache. Each cache has to use at least one dictionary. A high number of dictionaries may be ineffective.

**Size** – the size of each dictionary. All dictionaries will have the same size. Large dictionaries may be ineffective.

## Appendix D – Indexation statistics

Indexation statistics are created during the indexation process for each index component, in specified intervals. The statistics files contain the following columns:

**Processed items** – the total number of processed items (for all intervals).

**Tick count** – the environment tick value in milliseconds (for all intervals).

**Readings (R)** – the number of reading operations.

**Writings (W)** – the number of writing operations.

**File changed (FC)** – the number of operations that required the file to be changed.

**Nodes removed (NR)** – the number of nodes (inclusive leaves) that have been removed from the cache.

**Leafs removed (LR)** - the number of leaves that have been removed from the cache.

**Time required (TR)** – the time in seconds that has passed from the last statistics output.

**Reading var. (RV)** – the number of readings from the variable size block cache.

**Reading fix (RF)** – the number of readings from the fix size block cache.

**Reading key var. (RKV)** – the number of reading from the key variable size block cache.

**Writing var. (WV)** – the number of writings (insertions) to the variable size block cache.

**Writing fix (WF)** – the number of writings to the fix size block cache.

**Writing key var. (WKV)** – the number of writings to the key variable size block cache.

**Reading disk fix (RDF)** – the number of readings from the files that contain fix size blocks.

**Reading disk var. (RDV)** – the number of readings from the files that contain variable size blocks.

**Reading disk key var. (RDKV)** – the number of readings from the files that contain key variable size blocks.

**Writing disk fix (WDF)** – the number of writings to the files that contain fix size blocks.

**Writing disk var. (WDV)** – the number of writings to the files that contain variable size blocks.

**Writing disk key var. (WDKV)** – the number of writings to the files that contain key variable size blocks.

**Blocks split var. (BSV)** – the number of key variable or variable size blocks that have been split in order to fit into a simple or smart block.

**Block split fix (BSF)** - the number of fix size blocks that have been split in order to fit into a simple or smart block.

**Block cache flushed var. (BBFV)** – the number of times a dictionary from the variable size blocks cache has been flushed.

**Block cache flushed fix (BBFF)** – the number of times a dictionary from the fix size blocks cache has been flushed.

**Cleared references (ClearedRef)** – the number of references to a child node that have been removed. This is done when it is necessary to remove some nodes in order to maintain the predefined maximum total size.

**Items added to buckets (ItemsAddedTotBuckets)** – the number of items that have been added to the buckets.

**Items removed from the buckets (ItemsRemovedFromBuckets)** – the number of items that have been removed from the buckets.

**Full buckets (FullBuckets)** – the number of buckets that have been full.

**Full nodes (FullNodes)** – the number of nodes that have been full. Takes in consideration the number of items in the bucket.

**Buckets full failure (BucketFullFail)** – the total number of bucket insertions that have failed because the maximum buckets size has been reached.

**Existing items added (AddExistingItem)** – the number of items that have been added but their key was already present in the index.

**New items added (AddNewItem)** – the number of items that have been added and their key was not present in the index.

**Existing items added to a sub-tree (AddExistingItemSub)** – the number of items that have been added to a sub-tree but their key was already present in the sub-tree.

**New items added to a sub-tree (AddNewItemSub)** – the number of items that have been added to a sub-tree and their key was not present in the sub-tree.

**Nodes total size (CurrNodeSize)** – the total amount of bytes occupied by the B+-tree nodes.

**Number of buckets (NrBuckets)** – the total number of buckets.

**Buckets total size (CurrBucketSize)** - the total size that the buckets occupy.

**Indexed items size (IndexedItemsSize)** – the size in characters of the items that have been hashed and added to the index.

**Maximum data list size (MaxDataListSize)** – the maximum number of items in the data list from the B+-tree node leaves.

**Maximum data list size in sub-trees (MaxDataSubTreeListSize)** – the maximum number of items in the data list from the leaves of the sub-tree (second level).